UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

Rilevazione dei fallimenti nel sistema operativo *open source* Linux per applicazioni critiche

RELATORI
Ch.mo Prof. Stefano Russo
Ch.mo Prof. Domenico Cotroneo

CANDIDATO
Roberto Natella
matr. 885/99

ANNO ACCADEMICO 2006/2007

Contents

List of Figures

List of Tables

Listings

Introduzione

In un crescente numero di casi si fa ricorso a componenti software sviluppati da terze parti (COTS - Commercial Off-The-Shelf) anche per la realizzazione di sistemi critici per tempo ed affidabilità, per motivi di costo e di tempo. La necessità di rientrare nei vincoli di time-to-market, ossia di offrire sul mercato una soluzione prima dei concorrenti per acquisire un vantaggio, contrasta tuttavia con i lunghi tempi richiesti per sviluppare e collaudare un prodotto garantendo al contempo elevati livelli di qualità e di affidabilità. In particolare, sistemi operativi COTS non espressamente realizzati per operare in contesti critici sono presi in considerazione in tali casi; un esempio di questi sono i sistemi operativi open source, alla cui categoria appartiene Linux, il quale è stato ed è oggi impiegato per missioni spaziali, per il controllo del traffico aereo e ferroviario, per applicazioni bancarie, nei quali i malfunzionamenti rappresentano una minaccia all'incolumità di persone e cose, o possono comportare ingenti danni economici. In particolare, i guasti del software (nella accezione, ormai comune, di *bug*) rappresentano attualmente la maggiore causa dei fallimenti dei sistemi informatici.

Tipicamente i sistemi operativi COTS offrono dei meccanismi molto limitati per la rilevazione e la tolleranza dei fallimenti, ed inoltre essi non sono quantitativamente caratterizzati dal punto di vista della loro affidabilità. Gli approcci esistenti per garantire la affidabilità dei sistemi operativi (si veda il capitolo 1) si basano su una opportuna progettazione ex-novo (per esempio, i sistemi a microkernel quali Minix e QNX), oppure consentono la sola valutazione dei sistemi esistenti (ad esempio le tecniche come la Field Failure Data Analysis, Robustness Testing e Dependability Benchmarking), ma non permettono di incrementare l'affidabilità di sistemi operativi COTS già esistenti, che rimane ad oggi un problema ancora irrisolto. In tal senso, il seguente lavoro di tesi si propone di:

1. Valutare la capacità di rilevazione dei fallimenti del sistema operativo Linux (in termini di latenza e di copertura, intesa come probabilità di effettiva rilevazione di un guasto) utilizzando gli approcci esistenti per la valutazione della affidabilità di sistemi basati su componenti COTS; allo scopo, una specifica di dependability benchmark è stata formulata ed implementata (attraverso gli strumenti messi a disposizione dal sistema, e descritti nel capitolo 2). Il capitolo 3 discute approfonditamente gli aspetti legati alla valutazione dei sistemi attraverso l'introduzione artificiale di guasti, quali ad esempio la scelta dei guasti stessi, la quale è indispensabile per trarre dei risultati realistici sul comportamento che avrà il sistema in presenza di fallimenti.

2. Studiare una possibile tecnica per incrementare la capacità di rilevazione dei fallimenti; l'approccio considerato è basato sul monitoraggio on-line (ossia durante l'esecuzione) di grandezze significative che caratterizzino l'esecuzione del sistema (ad esempio, indici sintetici delle performance quali il numero medio per unità di tempo di operazioni elaborate dai driver dei dispositivi hardware), che permettano di comprendere se si è verificato un fallimento. Nel capitolo 4 il problema è analizzato quantitativamente, sulla base di osservazioni sperimentali, introducendo un algoritmo per effettuare la rilevazione.

3. Valutare sperimentalmente l'efficacia della tecnica di rilevazione proposta, rispetto ad una applicazione ed ad un modello dei fallimenti rappresentativo di contesti critici (si veda il capitolo 5).

Una efficace rilevazione dei fallimenti è mirata a consentire eventuali interventi sul sistema per recuperare un adeguato livello di corretto funzionamento, o almeno per limitare le conseguenze di un fallimento, in modo da incrementare l'affidabilità del sistema in oggetto. Un tema di ricerca attualmente rivolto in questa direzione è quello della *diagnosi del software*, che riguarda lo studio di tecniche per risalire alle cause di un fallimento di un sistema complesso durante la sua esecuzione, per poter predisporre il miglior tipo di trattamento possibile per il problema verificatosi; a questo scopo è indispensabile poter rilevare in maniera tempestiva e corretta la presenza di un fallimento, e fornire un supporto per una eventuale diagnosi dello stesso, che rappresentano i principali obiettivi di questo lavoro di tesi.

CHAPTER 1

Dependability of Operating Systems

> Most production software bugs are soft: they go away when you look at them.
>
> Jim Gray

1.1 Introduction

Nowadays, dependability is a big concern in computer systems engineering; in particular, software dependability is an important aspect of a complex system. Twenty years ago engineers realized that hardware was no more the main source of systems failures [SCK04] [Gra86]; instead, software became the prominent cause of system outages and malfunctioning, combined with environmental conditions and operator errors. The main reason is software complexity: modern systems have to fulfil many functional requirements, so they increased in size and scale; moreover, they were increasingly subjected to the time-to-market constraint that reduced design and testing efforts. This fact led to adoption of Commercial Off-The-Shelf (COTS) components, i.e. ready-to-use software that can integrated in a bigger system, in order to minimize its production cost and time.

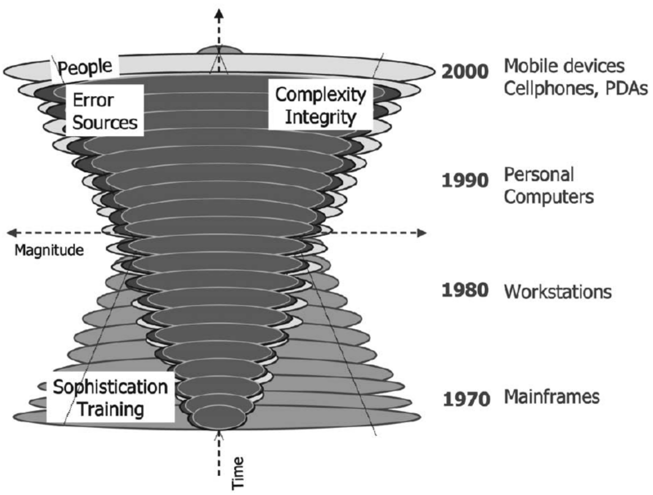


Figure 1.1: *Temporal trends for computer systems: there was a growth in complexity, integration, number and type of error sources; user basis also grew, but its sophistication and training decreased.*

Today COTS components are essential for any non-trivial software system; they range from operating systems (OS), compilers, middleware, virtual machines, web servers, to many kind of libraries and applications. COTS-based design introduces many issues for the design of dependable systems, because both their behavior in presence of faults and consequences of their interactions are unknown. Operating systems (to which we will refer in this document) are perhaps the most critical COTS components: they have to manage a broad spectrum of hardware components, and provide fundamental facilities to user applications like memory and storage management, scheduling, communication, networking. Moreover, their reliability affects the overall system's dependability attributes, therefore their characterization in respect of faults is prominent in the design of COTS-based systems. A way to characterize COTS components dependability is needed, that has to be efficient in respect of time and cost, unbiased, and quantitative.

1.2 Software Dependability

Software faults are a more recent, and elusive, class of failure sources for computer systems than hardware faults. While hardware faults are well coped by engineering practices in design and manufacturing, more work has to be made on software dependability; the former are commonly "physical" faults (e.g. an hardware component that wears out, or a broken solder), the latter instead are "human" faults (e.g. a mistake in the source code of a program, or in a system configuration file), i.e. they are introduced in the system during the development phase of its lifecycle [LRAL04]. Runtime mechanisms and architectures for achieving software fault tolerance where introduced, such as N-Version Programming, Recovery Blocks, and N-Self-Checking [Lyu95]; other approaches for fault prevention, removal and forecasting will be introduced in the following sections.

Classic software engineering practices like testing are not sufficient for fault-free software releases. Gray's hyphotesis [Gra86] states that most software faults are transient (*Heisenbugs*), i.e. their activation (and following manifestation) is non-deterministic because of software complexity: remember that in COTS-based layered software complex interactions between components and the environment take place, leading in occasional and non-reproducible failures, such as race conditions, deadlocks, and so on.

Transient faults are better described by a pair of attributes: *defect* (the meaning of the error in the source code, e.g. a wrong arithmetic operation, or an incorrectly assigned value) and *trigger* (the condition that allows a defect to manifest itself). Orthogonal Defect Classification (ODC) [CBC$^+$92] [Lyu96] is an affordable methodology that describes how to classify, by defect and trigger, software faults found during the development and testing phases, in order to improve the development and testing processes themselves (e.g. by focusing tests in a particular phase of software lifecycle) and produce reliable software. The ODC is commonly employed in several works on dependability to describe and classify software defects (e.g. studies on the field, fault injection methodologies). The choice of the classification categories should satisfy several conditions, in order to relate the defects to the development process phases (to produce a feedback for developers) and to make the classes disjoint and exhaustive at the same time (to keep the classification simple and unambiguous for developers). An allowed set of classes is described in table 1.1; other sets for defect types and triggers are reported in § 1.3.

1.3 Defects in Operating Systems

In this work, we consider dependability assessment and improvement of operating systems. In the past, several papers investigated software defects and their impact on operating systems' dependability, by analyzing data collected during testing and operational phases. [LI93] presents a measurement-based study of the Tandem GUARDIAN90 operating system; it analyzed software failures reports from all customer sites, together with actions taken by Tandem analysts in diagnosing and fixing the problems. The propagation and the effects of software faults on the system state where studied: faults are classified by the *underlying causes* (the design or coding fault in the source code), the *first error* after their activation (the immediate effect on the system state), their *propagation characteristics* (the error may propagate through the current task, across tasks or does not propagate at all), *error latency* (the time elapsed between activation and detection), and the *detection mechanism* (e.g. an exception triggered by an address violation, or an internal check). Moreover, an evaluation of software fault tolerance is given (Tandem's operating system implemented a process pair mechanism in which a failed process is replaced by a backup process on a different processor).

Software faults were the main failure cause (179 out of 200 reports). Fault causes where grouped in orthogonal categories, e.g. computation error, missing operation, and so on (see also [CBC$^+$92]); the most reported cause was "unexpected situation", that is cases in which potential operational situations were not considered by designers and the operating system can not handle them (e.g. race/timing problems, unexpected machine state or operational scenarios). Those faults were recognized mostly in more mature versions of the operating system or its components, instead new components are mostly affected by simpler fault causes; those results support the hypothesis of the presence of transient faults (Heisenbugs) in large systems, i.e. faults triggered by complex environmental conditions that are difficult to reproduce

Table 1.1: *General defect types adopted for Orthogonal Defect Classification (ODC).*

Defect type	**Description**
Algorithm	Efficiency or correctness problems that affect the task; fixed by reimplementing an algorithm or a data structure, without design changes
Assignment	A value is assigned incorrectly to a variable, or it is not assigned at all; affects a few lines (e.g. data structure initialization)
Checking	Missing or incorrect validation of parameters or data before their use (e.g. conditional statements)
Function	An error that affects an important functionality or feature of the system or a component; requires a formal design change and affects a sizeable amount of code
Interface	Communication problems between users, modules, components or device drivers
Timing/Serialization	Missing or incorrect management of shared and real time resources
Build/Package/Merge	Defects due to mistakes in library systems, management of changes or version control
Documentation	Defects in publications and maintenance notes

and debug. The high tolerance to software faults (82% of reported faults) is motivated by this behavior: the backup process has a different execution environment than the faulty process, in which software faults are not exposed to the same (transient) occurrences of random events, concurrent operations or memory state.

The error propagation is then discussed; reported faults have mostly a short error latency (the error was detected before the task that caused the first error was completed), and were quickly detected (the use of redundant data structures and consistency checks led to detection on the first access or by the first task that accessed an error in the memory state). Nevertheless, several faults presented a significant error latency, and they propagated across tasks (e.g. corruption of system global data or processes' private space): it was observed that errors sharing the same fault cause produced very different symptoms in the presence of depth error propagation. The early detection of the presence of a problem is recognized as a key requisite to reduce the variety of the symptoms of software failures and to simplify the automatic diagnosis of known problems, i.e. to identify the root cause of a failure and allow online recovery.

Another paper describing field failures in operating systems is [SC91]: software defects reported from an IBM operating system are classified by error type, trigger, symptoms and impact on the customer. The paper emphasizes *overlay errors* (i.e. defects leading to overlapping and corruption of memory areas) over the rest (*regular errors*): it is shown they have the highest impact, in terms of availability, perceived severity, and pervasiveness across the customer base. Tables 1.2 and 1.3 enumerate orthogonal categories in which reports were fit; checkmarks indicate defect types that caused overlays. Table 1.4 shows a classification for errors caused by software defects for the Tandem GUARDIAN90 operating system (proposed by [LI93] and [CC96]).

The most frequent overlay errors were allocation management, copying overrun and pointer management. It was possible to discover the typical characteristics of an overlay error: the majority of overlays corrupted a memory area no longer than 100 bytes, and they were close to the intended correct addresses; this information can be exploited both by recovery system design-

Table 1.2: *Defect types characterizing software defects of an IBM operating system [SC91]. Overlay-prone defects are those responsible of overlaps of memory areas, and they are a major part of the total.*

Defect type	**Description**	**Overlay-like**
Allocation management	A region of memory is still accessed after deallocation	✓
Copying Overrun	Overwriting beyond the end of a buffer	✓
Pointer Management	Address corruption	✓
Register Reused	Overwriting of processor registers	✓
Type Mismatch	Wrong implicit assumptions about a message format or structure	✓
Unitialized Pointer	Address not initialized	✓
Undefined State	The system goes an unanticipated state	✓
Unknown	Overlay error not classifiable	✓
Data Error	Algorithmic error	
PTF Compilation	Error in the bug-fixes distribution	
Sequence Error	Wrong order in messages or operations	
Statement Logic	Omission or disorder of statements	
Synchronization	Error in locking or synchronization code	✓
Deadlock	Tasks wait indefinitely because of resource locking	
Unclassified	Not categorized	

Table 1.3: *Defect triggers characterizing software defects of an IBM operating system [SC91].*

Defect trigger	**Description**
Boundary Conditions	Limit execution conditions: unusual parameters, hardware or system configuration, workload
Bug Fixes	Fixing of earlier errors
Client Code	User applications running in protected mode
Recovery/Error Handling	Execution of a recovery routine
Timing	Unanticipated sequence of events
Unknown	Not determined

ers (e.g. to bring off-line corrupted regions of memory before failures) and fault injection experiments. Whenever an overlay corrupts large and unrelated memory areas, this error is more prone to propagation and leads to severe failures.

Regular error types were concurrency-related, such as synchronization, sequence error, and undefined state; they often appeared in network and device protocols, and caused deadlock conditions and non-consistencies in the current state; their most frequent trigger was timing, e.g. untested interleaving of events. Surprisingly, boundary conditions triggered the majority of faults (both overlay and regular); again, unexpected conditions continue to arise once the software is released (e.g. software and hardware configurations, unusual input parameters, workload), therefore more intensive testing activities are needed to identify such problems. Recovery code is accounted as another frequent trigger, because it is difficult to test. It is also shown that bug-fixes themselves contained faults, often belonging to the Interface ODC category (e.g. type mismatch).

In [IR85], the relationship between software failures of the operating system and system workload is examined. It is shown that the operating system reliability is influenced by the type and nature of system activity: the analysis

Table 1.4: *Error categories for an IBM operating system [CC96].*

Error type	**Description**	**Subtypes**
Single address error	Incorrect address word	Control block address; storage pointer; module address; linking of data structures; register
Single non-address error	Incorrect non-address data	Value; parameter; flag; length; lock; index; name
Multiple errors	Combination of single errors or related to a data structure	Values; parameters; address and something else; flag and something else; data structure; random
Control errors	Memory not affected, or non-deterministic corruption	Program management; storage management; serialization; device management; user I/O; complex

of field crash failures and the workload prior to the occurrence of the failures points out that the *incremental risk* of a software failure (the ”hazard” as function of the workload level) increases exponentially with increasing workload; it is emphasized that the failure rate is not correlated with the total CPU usage (at its peak for the most of the time), but more complex factors contribute to failure manifestations. For the analyzed system, the leading workload variables were the operating system overhead, the paging rate and the average number of starting I/O operations per second, that represent the amount of interactive processing. The causes of such dependency were accounted to:

- the I/O activity, because an high number of exceptional I/O problems led to error handling failures;
- latent transient defects, because periods of stress or rare workload patterns lead to the execution of unused code, thus uncovering and activating more faults;
- the violation of space or timing constraints, because critical sequences of events were produced under extreme conditions only.

In [CC96], the data reported in [LI93] and [SC91] is discussed; the similarity between the statistical distributions of the error categories is emphasized (each error category has about the same occurrence probability in both the examined systems), giving a flavour of generality to the results; then the data is reused for injection of errors representative of software faults observed on the field.

1.4 Dependability evaluation

1.4.1 Experimental and analytical techniques

In order to build safety or business critical systems, appropriate techniques are required to make dependable design choices, to evaluate existing COTS components to integrate, and to assess dependability attributes of the final system; those techniques can be model-based or measurement-based, and are both applicable in different phases of the system life-cycle.

In model-based techniques, a formal analytical representation of the system is examined; examples of such models are reliability blocks, fault trees, and state-space models (e.g. Markov chains, Petri nets). Analytical models were mainly adopted in reliable hardware design, rather than in software, because of the explosion of model's complexity (e.g. the number of states representative of the behavior of the system), and different approaches were proposed for hierarchical specification of models for large component-made systems. Nevertheless, analytical models were successfully employed in design of critical software systems.

Strengths of such models lie in their support for theorem proving: after a system is modeled, an evaluation of its properties (e.g. liveness, safety) can be made by automatic tools. Moreover, dependability measures (e.g. availability, reliability) can be calculated: a model contains several parameters represented by stochastic processes (e.g. the probability that a transition between states is made, or that an event occurs), from which aggregated measures are derived. Modelling is applied both in early design phase (in order to choose between different fault-tolerant architectures) and prototipal and tuning phases: indeed, measurements from a running system are collected and elaborated in order to define numerical parameters of an analytical model; these measurements might come from an on-line production system (a previous release of the same system, or a system from the same family of products), or a prototipal version of the designed system.

In the case of production systems, on-line monitoring and data elaboration is conducted according to Field Failure Data Analysis (FFDA) techniques [XKI99] [SKS02] [SK05]: such information has to be collected in a large period of time (an order of magnitude of months, or even years) to obtain significant results, because dependable systems seldom fail. Information sources include standard system log facilities and dedicated monitoring probes. Field data truly reflects the behavior of the observed system, because they are collected in the real environment and inputs for which the system is conceived. Insights into classes of failures (with perceived severity) and their temporal distributions can be gained, and fault-tolerance mecha-

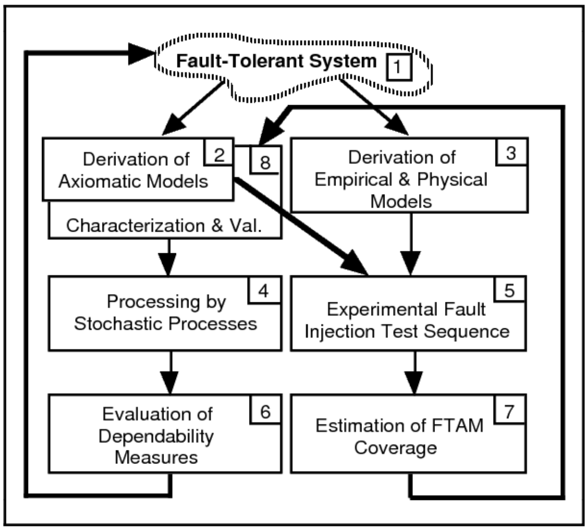

Fault-Tolerant System
1
Derivation of Axiomatic Models
2
8
Derivation of Empirical & Physical Models
3
Characterization & Val.
Processing by Stochastic Processes
4
Experimental Fault Injection Test Sequence
5
Evaluation of Dependability Measures
6
Estimation of FTAM Coverage
7

Figure 1.2: *Logical interactions between analytical and experimental evaluation techniques. A formal model can lead to definition of experiments, and collected results can feed numerical parameters of a model.*

nism can be evaluated: for example, the most common failure modes can be discovered, and error propagation in distributed systems can be observed by error correlation in different nodes.

Because of the large time needed, a FFDA campaign is more often used for long-term evaluation and tuning of an existing system than for designing a new one. Fault injection based experiments are conducted in order to shorten required time [ACC^{+}91], by artificial insertion of faults in a real system. Fault injection can be conducted during the validation of a system, or to assess dependability of existent (COTS) components; fault injection can also be simulated in a model (e.g. in computer-aided circuit design). Fault injection aims to:

- Study fault activation effects and their propagation in the system;
- Evaluate and improve fault tolerance mechanisms effectiveness (*coverage*, i.e. the probability of detecting an existing error, see [LRAL04]), that is fault forecasting and removal of fault tolerance deficiencies.

Both hardware and software faults can be injected; hardware faults can be simulated by injecting software faults that represent them (e.g. bit flips), namely software implemented fault injection (SWIFI). Software faults (i.e. defects in the source code) can be injected (or their activation can be emulated) by:

- Modifications in the original source code (mutation testing);
- Corruption of the memory contents [GKI04];
- Modifications in the executable code, i.e. mutation of machine-level code patterns with instructions representing high-level software faults [DM04] [DM02];
- Corruption of function parameters at API or driver level by bit-flip or selective substitution [AAF04] [KJA+04] [JAC+02].

A fault injection experiment is defined as a test sequence, where inputs belong to a set of injectable faults F and a set of activation profiles A (i.e. the workload), and the outputs consist of a set of readouts R collected from the system and a set of measures M derived from the analysis. Experiments type (a point in the $F \times A$ space) and number (limited because of time constraints) should be chosen with care, in order to produce non-biased results and extend them to the overall system (both tested and untested parts). A set of experiments should be designed accounting of:

- Representativity of the activation profile;
- Actual activation of injected faults;
- Probability of occurrence of injected faults in the system.

An operational profile, i.e. a statistical description of system usage, should be obtained in order to submit a representative activation profile [MA89] [Mus93]. If an operational profile is not available, the risk of bias can be compensated by submitting several activation profiles irrespective of injected faults and injection time.

In order to shorten experiments, [CC96] proposed an approach for injecting errors instead of faults, avoiding unactivated injection experiments and delays between fault injections and activations; the methodology is based on field data, and guarantees that injected errors only emulate software faults and reflect location and type of faults occurred in real systems. Moreover, the adoption of software engineering testing techniques is suggested to ensure the execution of paths in the code in which (emulated) faults are injected.

1.4.2 Robustness Testing

Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. Robustness testing is aimed to evaluate the ability of a component to resist and react to exceptional and erroneous inputs; for operating systems, these can be viewed as external faults coming from the system calls, the driver interface or from the hardware layer directly.

Robustness testing was introduced as an automatic approach for validation of exception handling and error notification mechanisms of a system. This approach does not require any in-depth knowledge of the system in hand (which represents a ”black-box”), its source code or a behavioral specification, but only its interfaces description; for this reason, robustness testing is valuable for COTS components evaluation. Quantitative measures can be assessed for an OS (and its workload) about its behavior and performances in the presence of faults in external components. These experiments aim to discover robustness failures of the target component, that is unexpected (but yet possible) inputs that are not correctly managed; it is worth noting that robustness failures do not necessarily correspond to software faults in the considered component (e.g. the occurrence of a crashing input is not considered by the component specification and never used, ever if external components are faulty), but they should be considered as potential vulnerabilities.

An example of tool for automated robustness testing is Ballista [DKG99]: it can generate test cases for system calls of POSIX compliant systems, based on their signatures; malformed invocations are created by combination of input values from a common knowledge base, that contains ”critical” values for each data type (e.g. the NULL value for a pointer in the C language). Tests can be extended include non-standard and user-defined data types.

Figure 1.3 shows results from robustness testing of different releases of several operating systems. The robustness failure rate (the percentage of non-properly handled erroneous inputs) is one of possible measures that can be obtained; results can be interpreted after classifying robustness failures by the relative importance attributed to different observed behaviors (e.g. workload abort, silent failure, and so on). The robustness failure rate typically decreases between releases if upgrades incorporate reliability improvements, and increases if new features are introduced without sufficient testing.

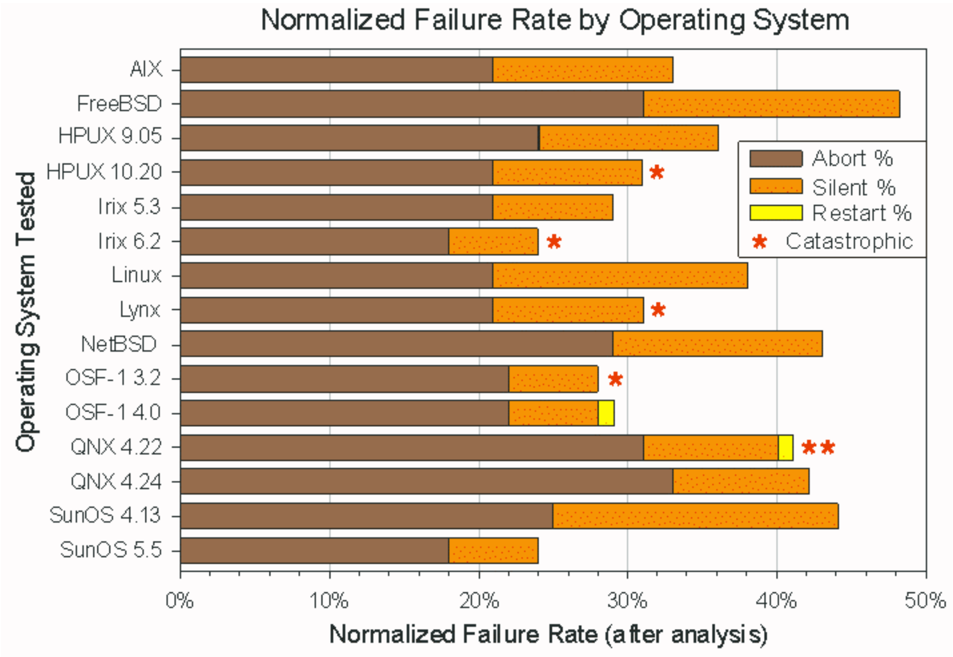


Figure 1.3: *Results of robustness testing at the system call interface for 15 POSIX compliant operating systems.*

After robustness testing, several unproperly handled inputs are identified. The exception handling mechanisms can be improved by using a software wrapper [DK02] [PKS$^+$01], i.e. an interface to the COTS component that screens out exceptional parameters for specific functions, providing graceful failure notification for known robustness failures' inputs: an error code is returned, or an exception is thrown, and the service request is not executed at all by the COTS component. Robustness testing results can be used to improve test suites, and to better understand exceptional conditions that trigger robustness failures.

1.4.3 Dependability Benchmarking

Dependability benchmarks are a generic way to characterize a component or a system in the presence of faults, allowing to quantify dependability attributes. A dependability benchmark describes a procedure to obtain such measures, that verifies the following standardized properties [MKA$^+$01] [DM04]:

Representativeness : the workload and faultload submitted are those typically experienced by the benchmarked system;

Portability : in order to compare a set of systems in a given category, the benchmark has to be implementable for all of them;

Repeatable : results produced should be (statistically) reproducible across repeated benchmarks;

Feasibility : the time and the efforts needed to implement and execute the benchmark are minimal;

Low intrusiveness : the perturbation in the system by the benchmark implementation should not alter the meaning of the results.

Dependability benchmarks standardize common dependability evaluation techniques in an unambiguous and clear manner, and simplify interpretation, understanding and comparison of results; they are similar in intents to performance benchmarks, shifting the scope to dependability. Dependability benchmarks already exist for general purpose operating systems, On-Line Transaction Processing (OLTP) environments, real time and embedded systems.

The subject of the benchmark, the Benchmark Target (BT), is a subsystem or a component that is part of a larger system, the System Under Benchmark (SUB); the SUB includes all is needed to execute the workload and the benchmark itself. A faultload is the set of faults and exceptional conditions (including stressful workload) that are introduced in the SUB to emulate the behavior of a faulty system; experiments based on fault injection are conducted to assess the dependability of the BT. In order to keep the results meaningful, the BT is not directly affected by injected faults; faults are instead injected in the Fault Injection Target (FIT), another component that belongs to the SUB, and the impact of error propagation to the BT is evaluated. The FIT is chosen ensuring that its misbehavior will reflect on the BT, with an high fault activation rate (see figure 1.4).

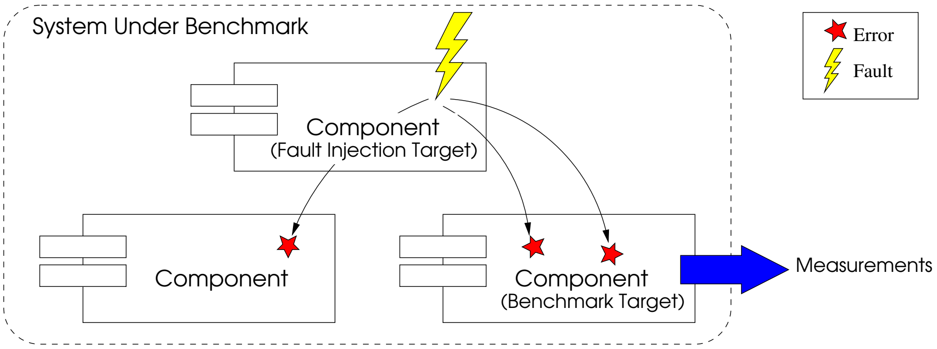


Figure 1.4: *The fault injection target of a dependability benchmark is the software component designated to spread errors to the benchmark target.*

The choice of the FIT is fundamental to pursue experiment effectiveness. A possible choice of the FIT for dependability benchmarking of an operating system (figure 1.5) are its drivers, that represent a significant part of its code, and are frequently reported as source of failures. If drivers have to be included in the BT, faults can be injected at hardware level (at least emulated by software, taking care of not modify the BT code), or at the userspace application level (e.g. the Ballista project, § 1.4.2). The operating system can also act as FIT if the BT is an userspace application, for example a web server.

It is worth mentioning the European Dependability Benchmarking project [MKA$^+$01], which defined a conceptual framework for specification of dependability benchmarks; it introduced several dimensions by which a benchmark can be described, grouped in three classes (figure 1.6):

Categorization : describe the targeted systems, the benchmark context and purposes;

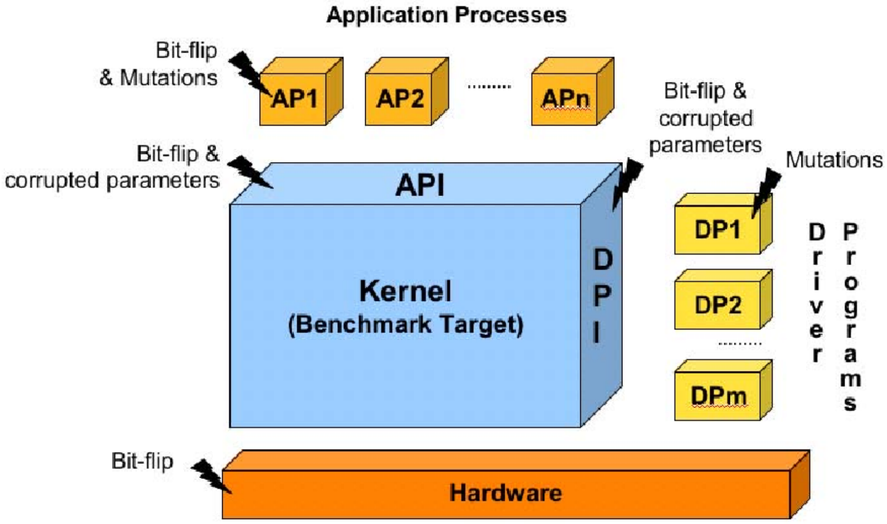


Figure 1.5: A perspective of dependability benchmarking from the operating system's point of view. Faults can be injected both in external components (device drivers and application programs), or at the interfaces between them and the operating system.

Measure : the set of attributes assessed by the dependability benchmark, in well-specified operative conditions, concerning the benchmarking context;

Experimentation : aspects related to the experimentation to obtain selected measures.

A measure may concern with the overall system at service delivery level (*comprehensive measures*, e.g. its availability) or a particular facet (*specific measures*, e.g. its ability to detect the presence of errors). Measures can be obtained directly from experimentation (namely, measurements) or from an analytical model parametrized with experimental readings (e.g. to calculate availability using a parametrized Petri net). Because fault activation may lead to a performance degradation in comparison of non-faulty scenarios, performances in the presence of faults should be measured.

In dependability benchmarks of general purpose operating systems (the BT), faults' impact on both SUB (the operating system with libraries and hardware) and the workload could be evaluated; two classes of measurements,

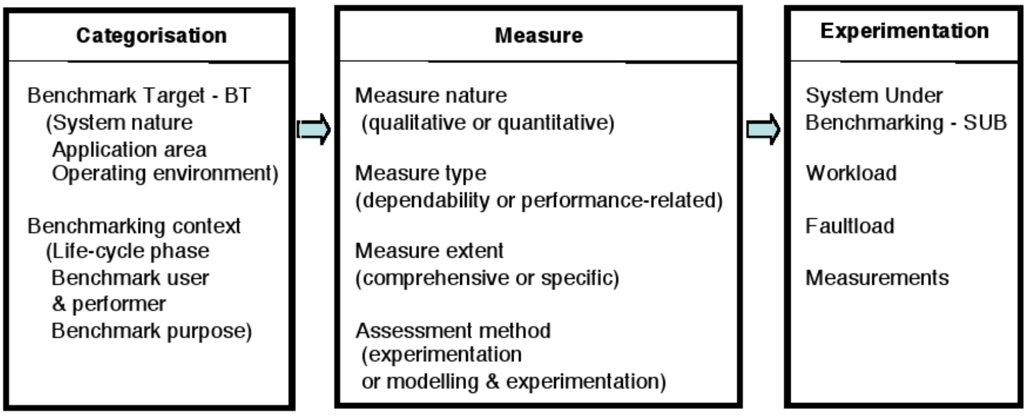
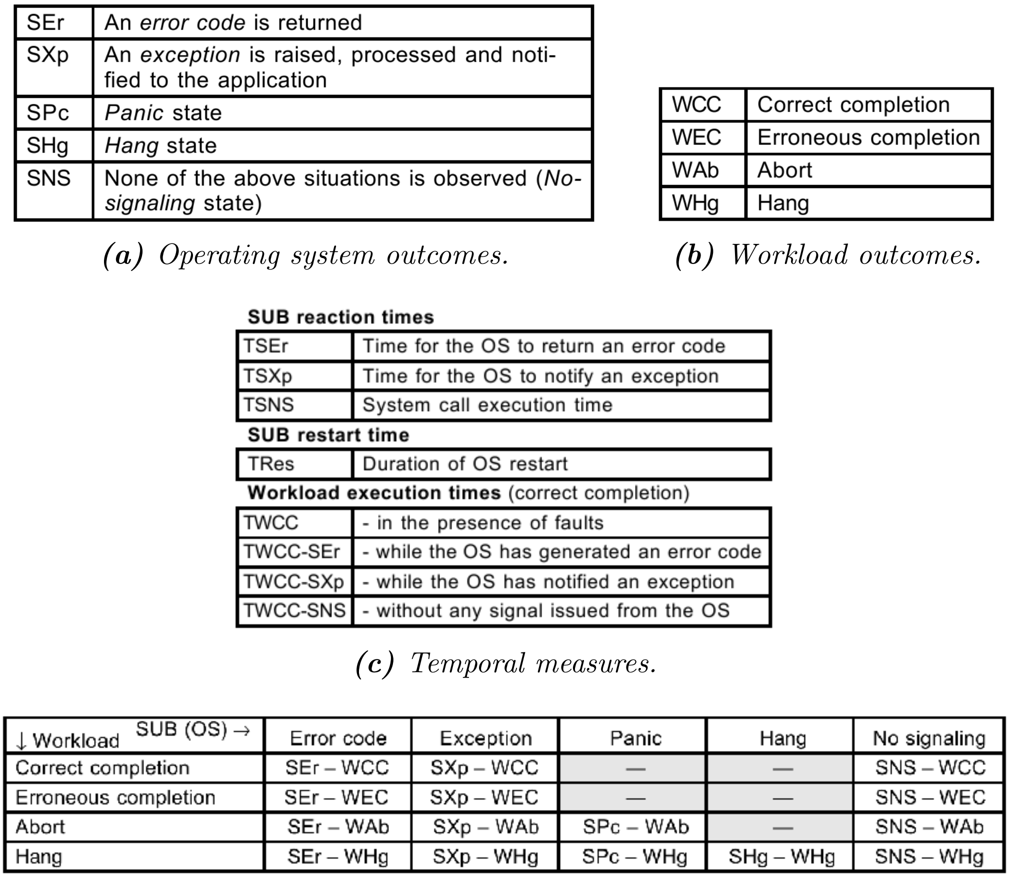


Figure 1.6: *Dependability benchmarking classes of dimensions.*

namely robustness and temporal, can be defined. Figure 1.7 shows possible experimental outcomes when faults are injected by system call parameters corruption.

It should be noted that although several measurements refer to the whole SUB (e.g. execution and restart times are hardware-dependent), they can also be considered as characterizing the BT (e.g. the absolute execution times are of little interest; instead, the difference between times in the presence of faults and in the nominal behavior may be considered).

Outcomes in figure 1.7a are a set of general high-level results from faulty execution of an operating system. When SEr and SXp occur, the workload is notified about the problem by an error code returned or an exception issued by the OS (in turn, the OS was triggered by an event such as an interrupt or a CPU exception). When an exception is issued during execution of kernel code, the OS enters in the *panic* state (the SPc outcome), where the workload is aborted; a soft reboot may be still possible. In the *hang* state (the SHg outcome), the kernel no longer replies to issued requests (e.g. it is executing an infinite loop, or waiting for an event when interrupts are disabled); an hard reboot is required. When the SNS outcome occurs, the kernel simply returns a result to the workload after the system call invocation (the result can be correct or wrong), without reporting errors; because the workload is still executed, any of the workload outcomes in figure 1.7b can occur. Observation of the workload final state is helpful in evaluating the impact of



SEr	An *error code* is returned
SXp	An *exception* is raised, processed and notified to the application
SPc	*Panic* state
SHg	*Hang* state
SNS	None of the above situations is observed (*No-signaling* state)

(a) Operating system outcomes.

WCC	Correct completion
WEC	Erroneous completion
WAb	Abort
WHg	Hang

(b) Workload outcomes.

SUB reaction times

TSEr	Time for the OS to return an error code
TSXp	Time for the OS to notify an exception
TSNS	System call execution time

SUB restart time

TRes	Duration of OS restart

Workload execution times (correct completion)

TWCC	- in the presence of faults
TWCC-SEr	- while the OS has generated an error code
TWCC-SXp	- while the OS has notified an exception
TWCC-SNS	- without any signal issued from the OS

(c) Temporal measures.

↓ Workload SUB (OS) →	Error code	Exception	Panic	Hang	No signaling
Correct completion	SEr – WCC	SXp – WCC	—	—	SNS – WCC
Erroneous completion	SEr – WEC	SXp – WEC	—	—	SNS – WEC
Abort	SEr – WAb	SXp – WAb	SPc – WAb	—	SNS – WAb
Hang	SEr – WHg	SXp – WHg	SPc – WHg	SHg – WHg	SNS – WHg

(d) Combined outcomes.

Figure 1.7: *Possible measurements for operating systems dependability benchmarking.*

the OS on the workload. The set of possible combined outcomes is depicted in figure 1.7d.

Other possible outcomes from the OS can concern to: the boot procedure (e.g. completed without warnings; not completed with or without failure informations); the OS ability to automatically reboot (during the boot procedure or workload execution); effects on the hardware (e.g. a device is disabled, unusable, re-initialized, or introduces errors on the data written or read by the OS); error codes returned by internal kernel function invocations (e.g. procedure calls made by device drivers to kernel facilities). Outcomes can be observed by internal assertions implemented in the kernel (e.g. by measuring the memory load - the average number of allocation requests -, the number of allocation requests that enlarge the allocated memory data segment, and other internal activities of kernel components). Those and other outcomes can be introduced in the benchmark specification; they can be derived by user reports or static code inspection, although the fault injection experiments can point out unexpected failure modes.

Temporal measures (figure 1.7c) include OS reaction times, i.e. the time elapsed before a feedback is provided (SEr, SXp, SNS), and OS restart time, i.e. the time needed for a system restart after a fault (e.g. time spent in recovery checks and actions); the latter can significantly affect the availability of the system. Workload execution times can also be influenced by a fault, and can be included between experimental readings. Temporal measures should be compared with ones in the nominal behavior; such nominal times should be measured for the instrumented kernel when instrumentation has no actual effects on execution.

Combined outcomes in figure 1.7d are an example of generic failure modes. However, failure modes can be related to a specific observation goal, depending of the particular context in which the OS will be integrated. Failure modes can be coalesced in fewer failure classes, and a severity can be assigned to each class according to the desired interpretation of benchmark results. Typical profiles by which failure modes can be grouped and ordered are:

Availability : failure classes in which the system can be considered still usable are preferred over that in which OS or workload availability are affected (e.g. no signalling is somewhat better than a crash);

Responsiveness : failure modes in which a feedback is returned to the user are preferred (e.g. a crash is better than no signalling);

Safety : the system is considered safe if its execution is correct even in the presence of faults (e.g. workload incorrect completion is worse than workload fail-safe stop by an exception thrown).

More than one outcome can be observed during an experiment. In the case of several outcomes, and experiments have to be categorized univocally by a specific outcome (e.g. the benchmark purpose is to measure the percentages of OS failure modes), the classification can be made by choosing an outcome among the set applying a criterion (e.g. the outcome severity, or the temporal order of events); this criterion has to be defined on the base of benchmark purposes, because it can affect the interpretation of results.

After the execution of a dependability benchmark, in order to guarantee reliable results, a verification of the benchmark's properties fulfillment (representative, repeatable) should be conducted. The faultload representativeness come from the choice of a fault injection technique which emulates real faults from the field (i.e. equivalence of the impact and consequences of faults, in the error domain) [JAC$^+$02]; moreover, if a limited part of the FIT is subjected to fault injection (e.g. a subset of the system calls is chosen for parameter corruption), a sensitivity analysis should be conducted in order to validate this choice (e.g. obtained results should be statistically similar if we vary the system call or data type corrupted, or the corruption technique). Representativeness of workload selection is also important, because it implies the activation of residual real faults, and therefore it is a prerequisite to assume the results as representative of the real operational system.

Results from the dependability benchmarking include the experimental outcomes after injection of several faults, that differ in type and location. From these results, a statistical characterization of the BT can be outlined; examples of conclusions that can be drawn by a dependability benchmark are:

- The frequency of each failure mode, and an estimation of the relative occurrence of these outcomes on the real system (if the faultload and the workload reflect field data);

- A qualitative characterization of the OS by a specific failure profile (e.g. by the types of failure modes occurred, the OS can be evaluated from the point of view of safety, availability, responsiveness);
- A comparison between several releases from the same OS family;
- An estimation of the robustness of the OS in the presence of faulty components (e.g. device drivers, applications);
- An estimation of the fault activation rate (by the ratio of the number of experiments in which a failure is observed to the total number of experiments);
- The extent of error propagation between kernel components (by comparing the fault-injected kernel component to the component in which a failure is signalled);
- The impact of faults on execution and restart times.

In order to improve the diagnosis of the impact of an injected fault, the subsequent execution of a "replay" workload can be made. This is useful to increase the fault activation rate and reduce the number of experiments without observed effects (i.e. no signalling); the replay workload can consist of the same workload used for the experiment.

Moreover, the system's current state can influence an experiment altering benchmark outcomes. However, this factor is not taken in account in benchmark specifications, in order to make it feasible: usually, the system is rebooted before the beginning of each experiment. Ramp up and ramp down phases, for each experiment, can be planned to increase and decrease gradually the system's load before and after a fault is injected, respectively, to establish the system's state, and experiments can be executed consecutively without reboots [DM04] (except for hard failures such as kernel hangs or crashes) if a sufficient time between them is waited (this wait should be much longer than the time required for former injected fault activation), and rebooting periodically in order to take in account FIT internal degradation; this procedure is somewhat similar to performance benchmarks. In general, fault injection techniques do not consider the effect of software aging [LRAL04], that progressively corrupts the system's state leading to failures: only fault analysis based on field data succeeds in detection of such phenomena [XKI99]. This problem is similar to the software engineering's issues in object-oriented code testing, in which the internal state of an object has to be taken in account to test its methods.

1.5 Dependability improvement

1.5.1 Improvement of COTS-based systems

The techniques described in previous sections focus on software dependability evaluation: they can be used in order to compare different systems or different releases of the same one, to discover the overall behavior of a (COTS-based) system in the presence of faults, and to estimate its adherence to dependability requirements. Those techniques can be applied both on the final system and on prototypes and models during development phases. Instead, existing dependability improvement techniques can only be applied on the latter, intervening on the development process: ODC [CBC^{+}92] and software-reliability engineering techniques [MA89] are applied to the development process in order to ship systems guaranteeing quantitative dependability attributes (by tuning duration and focus of testing and development efforts); moreover, run-time software fault-tolerance can be introduced in the system design phase to cope with software transient faults that unavoidably elude tests.

Such techniques can not be employed to increase dependability of already existing COTS components, that often are not designed with dependability in mind. Moreover, due to complexity, or to the lacking of source code and documentation, COTS integration can lead to subtle faulty interactions within the system [MP05]. We would like a technique for improving dependability of a complex, component-based system with the following features:

- Can be (semi-)automatically deployed and configured for an existing system;

- Does not require any in-depth knowledge of internal architecture of the system;
- Portable among a broad class of systems, at least of the same category (e.g. operating systems, web servers, database management systems);
- Feasible and cost-effective;
- Does not impose an excessive performance overhead.

1.5.2 System reengineering

Many works focus on reliable design of operating systems, e.g. microkernel based. In [HBG$^+$07], an operating system design (implemented in the Minix operating system [15]) is proposed that enables recovery from malfunctioning components: such design exploits isolation (by enforcing a privilege-based access scheme to logical and hardware resources) between the kernel and failed device drivers (running in user-level processes), to prevent error propagation and to tolerate failures by dynamic restart of a failed process: often the problem does not occur again after the replacement of a process, because the faults activation conditions (trigger) are occasional (transient faults). This approach relies on the ability of detecting a failure in monitored components (at the moment it is limited to fail-stop failures, e.g. a crash), therefore underlining the importance of an effective detection mechanism. In this work it is also suggested that the driver isolation idea can be generalized to monolithic commodity operating systems by enclosing them in user-level processes.

Other approaches to the isolation of faulty components are based on *virtual machines*, i.e. a fictitious execution environment in which applications and operating systems (guests) can run as they would on a private physical machine, being unaware of other concurrent virtual machines. Virtual machines usually come in the form of *full virtualization* and *paravirtualization*: in the former case, the virtual machine consists of virtual devices and

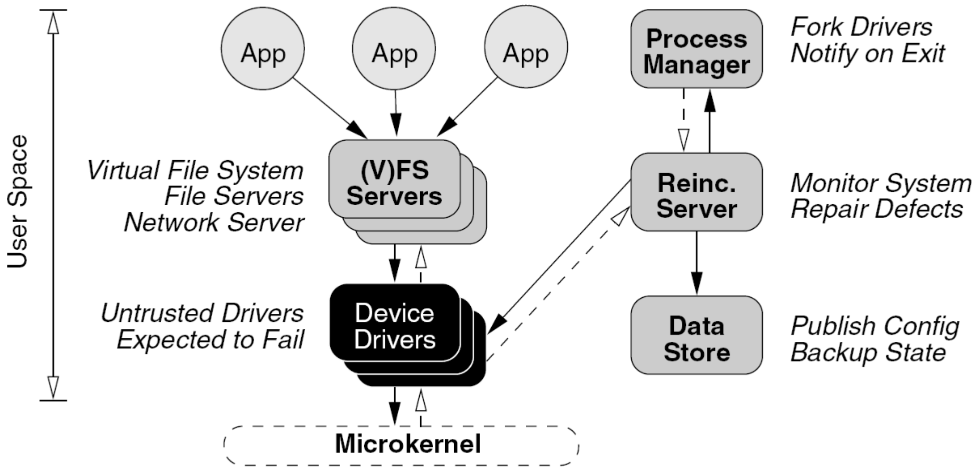


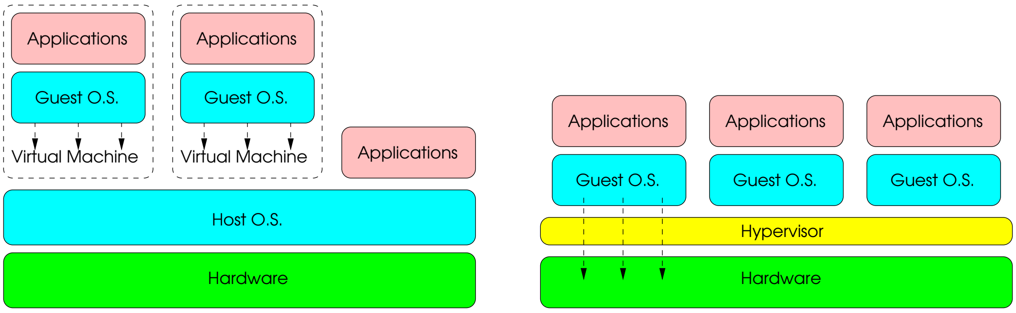
Figure 1.8: *The microkernel-based architecture of the Minix failure-resilient operating system [HBG+ 07]. Both device drivers and major subsystems run in isolated user-space processes.*

it executes as a process in the context of a real (host) operating system[1] [22] [20]; for example, a virtual disk may be implemented by allocating blocks on the filesystem of the host system; the process encapsulating the virtual machine will translate accesses to I/O addresses in requests to the host. In the latter case, the virtual machine (composed by an operating system and applications) runs on the real machine in the presence of a minimal software monitor, namely *hypervisor*, that manages the correct concurrent execution of virtual machines[2] [23]. See [21] for a full classification of the approaches to virtualization.

Virtualization is commonly recognized as an effective mean to increase the dependability by enforcing isolation between applications in different virtual machines. In [LUSG04], an approach is proposed to improve the dependability within a single operating system through isolation of device drivers in

[1] In general, the term "full virtualization" embraces all virtualization techniques that enables the execution of unmodified software on a virtual machine; because existing hardware does not allow the concurrent execution of multiple virtual machines, full virtualization can only be achieved by the complete emulation of hardware devices. Newer architectures support full virtualization of physical hardware [AJM+06].

[2] Because virtual machines delegate resources allocation to the hypervisor, their operating systems are partially aware of virtualization and require several modifications.



(a) *Full virtualization: virtual machines execute as processes on a host system; the hardware is fully emulated and processes and guest OS are unaware of virtualization.*

(b) *Paravirtualization: virtual machines execute on top of an hypervisor, that manages concurrent accesses to the physical hardware.*

Figure 1.9: *Approaches to virtualization.*

separated virtual machines: each virtual machine contains a subset of device drivers, and a failure in a virtual machine can not affect the other ones. The L^4 microkernel, which is similar to hypervisor, allows the virtual machine running the main copy of the operating system to communicate with other virtual machines running device drivers, and prevents interferences between virtual machines. This approach allows the reuse of the code of the device drivers, but still requires several modifications in the rest of operating system (both the main copy and the ones running drivers). The implementation of a case study based on Linux (L^4Linux) showed a performance overhead between 3% and 8%, with a code reuse ratio of 91% and about 10K of LoC (Lines of Code) to adapt the Linux kernel.

A further approach consists in the robust development of device drivers based on language-level protection. The Devil language [RM01] is an IDL (Interface Definition Language) for the high-level specification of the communication protocol between the driver and the device; such description is then used to automatically generate stubs in C language that can be used in the driver, avoiding the programmer to write error-prone code. Moreover, driver specifications can be checked for consistency during their compilation, and assertions are included in generated stubs code in order to check for correct

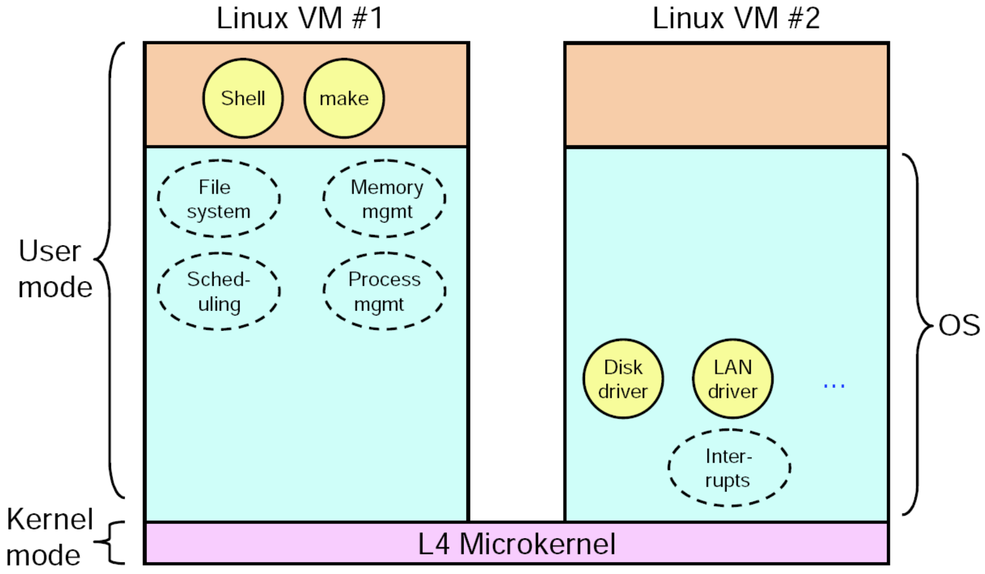


Figure 1.10: *Execution of Linux device drivers in isolated virtual machines (L^{4}Linux) [THB06].*

usage by the driver at compile time, and for correct (specification-compliant) behavior of the driver and the device at run time, thus preventing common software defects.

A similar language-based approach is employed in the design of the Singularity operating system [HL07]. The communication between programs (executed in SIPs, Software-Isolated Processes, similar to ordinary processes in other operating systems) is based exclusively on message passing through the abstraction of *contract-based channels*. In fact, applications and operating system's components are written in Sing#, an high-level and type-safe programming language, in which the communication is described in terms of a state-based protocol (like a Finite State Machine), using dedicated language constructs; static program verification is made both when intermediate bytecode (a set of CPU-independent instructions interpreted at run-time) and native code (CPU-dependent) are generated, to avoid that a non contract-compliant program could be loaded. Moreover, the Sing# language provides automatic memory management and garbage collection, and seals each SIP address space by other SIPs (e.g. by omitting pointers or similar constructs):

because each SIP is language-constrained to not access external memory areas, all SIPs can be executed within the same address space (communicating only with message exchanges), thus ensuring a low-overhead software isolation between processes (opposite to MMU-based isolation present in other operating systems, that requires to set-up further data structures at each context switch). Both user applications and operating system's components (e.g. filesystems, drivers) are developed in Sing# (thus dropping backward driver and application compatibility), and executed in the context of the Singularity microkernel. Finally, SIPs can not dynamically load or modify executable code, therefore code extensions (that are considered as a major source of failures) have to be executed in distinct and isolated processes; a *manifest* is needed to execute a program, describing the expected behavior, its required system resources, capabilities and dependencies, that are statically and dynamically verified.

1.5.3 Component wrapping

Component wrapping provides a way to wrap software interfaces to handle exceptional inputs and prevent failures; the Ballista project [DKG99] proposed a procedure to automatically discover robustness failures at the system call interface of POSIX operating system, based on the specification of function signatures and exceptional parameters' values (1.4.2). Another approach to increase operating systems reliability is to apply component wrapping to device drivers (and, optionally, to other kernel subcomponents); in the Nooks model [SBL03], each wrapper monitors interactions between a device driver and the kernel (figure 1.11). When the driver makes a call to a kernel function, or the kernel makes a call to a driver function, the correctness of exchanged parameters is first verified by the wrapper, and then the call is serviced: therefore developers have to explicitly write wrappers for semantic checking. Virtual memory management is used to protect kernel data from direct write accesses by wrapped drivers; a kernel object can be accessed only through wrappers, that copy data into temporary write-enabled location, preventing data corruption by a misbehaving driver. Nevertheless, drivers still execute in privileged mode and wrappers can not prevent malicious drivers from re-enabling write access to kernel data. Automatic recovery is managed by an user-space agent; when a failure is detected (through parameter checks, process exceptions or explicit external signals), a complete reboot of a device driver is made and resources are released; a transparent recovery to user-space processes is possible, but it is not guaranteed.

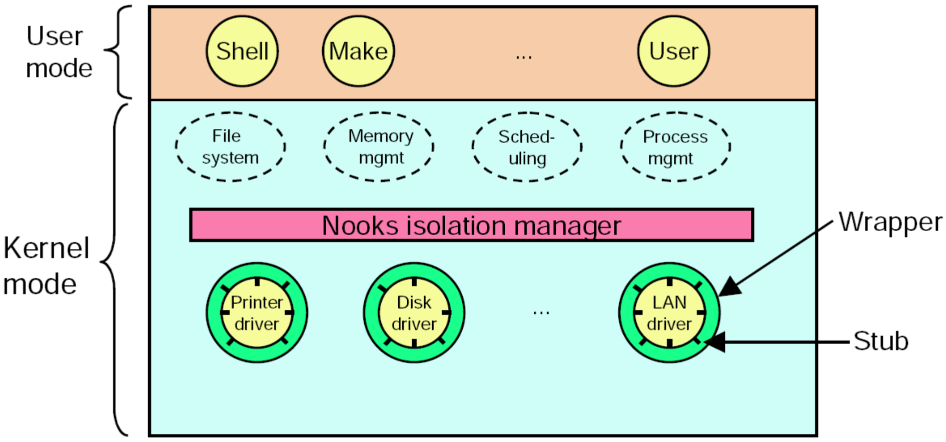


Figure 1.11: *The Nooks approach for device drivers wrapping, in order to check all values exchanged with the kernel for validity [THB06].*

1.5.4 On-line monitoring

Previous approaches (driver isolation, virtualization, language-based protection) enhance the operating system ability to isolate errors and allow recovery procedures; however, they need engineering efforts in order to adapt the system to a new architecture, and in some cases they require a rewrite from scratch of the operating system. Component wrapping turns out to be a more conservative approach, allowing for the reuse of a major part of existing operating systems' code; nevertheless, this approach can only work around known robustness failures discovered by exhaustive variable-inputs trials, and does not take in account more complex faults due to long-term execution and internal interactions, thus it leaves residual transient bugs in the source code. The data from the field shows that the most frequent error triggers were boundary conditions, recovery and timing; the execution on the operational site reproduces these triggers better than testing efforts, and so this approach can not discover most complex faults; moreover, the effect of software aging, by which the memory state is slowly corrupted (e.g. memory leaks, arithmetic approximations, resource exhaustion), is not considered. FFDA campaigns showed the extent of such problems; nevertheless, data is analyzed much time after collection. A complete technique should be designed to face unexpected faults, by discriminating a correct behavior of the system from a faulty one; this discrimination can be done by on-line monitoring and analysis of the system, i.e. the observation of critical variables during its execution. On-line *fault diagnosis* an innovative approach that aims to identify the presence of an error (*detection*) in real time, and to trace its causes in terms of location and type (*isolation*) in a COST or component-based system (the operating system is viewed as a part in a more complex design that includes middleware, third-party libraries and applications). After that a failure is traced, recovery and reconfiguration may follow to restore correct execution conditions [BCCR04].

As a consequence, on-line diagnosis is founded on the ability to notice a misbehavior of the system within a short period of time. Commonly, modern operating systems comprise several facilities for *event logging*: logged events may concern to regular activities on the systems, or they may be manifestations of internal errors; FFDA campaigns are often based on logging activities. However, if the operating system is not designed for dependability, its ability to log events of interest is limited: failure symptoms can go unnoticed or be misleading (low detection quality), or they are logged only after a significant amount of time (latency).

Another approach to detection consists in synthesizing the correct execution profile of a system in the form of invariants or rules, i.e. predicates that hold when the system behaves correctly, that can be checked subsequently. The difficult in defining a behavioral model that describes the correct execution makes this approach less attractive for application in large systems like operating systems: invariants have to be so general to include behaviors similar but not identical to expected ones, and also enough specific and accurate in such a way that their violation can be indicative of the causes of an error. Moreover, they do not take in account complex phenomena like software aging.

We aim to introduce a dedicated component, namely a *failure detector*, that produces a notification about the problem occurring when the system's state manifests symptoms of known faults and deviates from the correct behavior. Knowledge about faults and symptoms (a fault model) should be gained by pre-operational monitoring of the system (e.g. during fault injection and workload stressing experiments), to be employed for on-line monitoring by a failure detector later in the operational phase; the fault model should be incrementally increased, in order to improve quality of detection. The failure detector should also offer support to fault isolation (by generating useful hints about a fault for the fault diagnoser), have a small computational overhead, and provide a better Quality of Service (QoS) than standard logging facilities, in terms of accuracy and latency (QoS metrics for failure detectors are discussed in [CTA02]).

In the following chapters we are going to define and evaluate an approach for failure detection on the Linux operating system. Kernel execution will be monitored by means of internal probing; in the pre-operational phase, an analysis of system behavior under errors is made, that can be used by a failure detector in order to do on-line monitoring of the operating system; finally, its effectiveness in terms of accuracy and speed will be evaluated by conducting a case study on a distributed system with dependability requirements. We expect that such a failure detector can be useful for a more complex user-space agent responsible to autonomously correct the system behavior by applying recovery means, and thus leading to an improvement of overall dependability.

1.6 Conclusions

In the past years, different ways to evaluate and compare dependability of complex systems were proposed. Those approaches can be grouped in three categories: analytical modelling, field monitoring and analysis, fault injection experiments. Model-based methodologies are well established, and they provide useful qualitative insights and complex quantitative measures of system's behavior; nevertheless, the correspondence between the model and the real system is hard to be ensured, and its parameters have to be refined by experimental campaigns, therefore those methodologies seldom can be afforded for non-critical systems. More cost-effective techniques are based on actual execution and examination of system's behavior; they require little or no knowledge of system internals (”black-box“), and can be applied for large and distributed systems too. Techniques based on field data (FFDA) best extrapolate the dependability behavior in operational contexts (real phenomena like failure modes can be observed), but they require a large observation time and therefore not much can be done to improve the system itself.

Fault injection techniques are good for time and cost effective dependability evaluation; they can be applied in post-design life phases of software systems, and can be used for improvement of fault tolerance mechanisms. In order to improve effectiveness and portability, fault injection based frameworks were defined (namely robustness testing and dependability benchmarking), standardizing procedures for dependability evaluation and lowering time and efforts required to perform the experiments. Depending on user's characterization goal, a benchmark could be defined to best describe the system's behavior from a particular point of view (e.g. defining and classifying experiments' outcomes by availability, responsiveness and safety). However, workload and faultload representativeness are a critical factor in assessing dependability attributes, and do not take in account the system's state preceding a real fault; field data are essential if a comprehensive view on system's dependability is desired.

Finally, even if dependability evaluation can give a feedback to system's developers, it is of little help in defining strategies for automatic fault treatment and recovery. An approach is needed to improve the dependability of existing systems in a cost and time effective way; in fact, existing approaches, like microkernel design, didn't make a success because they are complex or unfeasible to maintain. A mechanism based on a failure detector will be introduced in the following chapters, that can be used to apply dependability means to COTS-based systems.

CHAPTER 2

An overview of Linux kernel analysis tools

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?
>
> Brian Kernighan

2.1 Introduction

In this chapter, we are going to review several utilities to inspect the Linux operating system; their goal is to collect field data from a kernel running in a test or production environment, in order to discover failure modes and to apply dependability means. Data of interest about the kernel state is its internal variables (running tasks, loaded modules, acquired locks, and so on), user and kernel stacks, and more in general the memory raw image during and after a failure.

The tools that will be covered are officially supported by Linux kernel developers, and included in the main kernel 2.6 tree; they also require several user space utilities, usually shipped with Linux distributions or freely available on the Internet. Other tools are referenced at the end of this chapter.

2.2 Kexec and Kdump

The Kexec patch is included in the Linux kernel since version 2.6.13. The `kexec()` system call reboots on the fly the running kernel, and executes a new kernel previously loaded in memory. It differs from the classical `sys_reboot()` call because the reboot does not need to go through hardware reset and BIOS startup, but directly executes the pre-loaded kernel image; only the last boot stage is actually executed, in which the kernel initializes its data structures, probes the hardware devices and load device drivers.

The reboot procedure can be triggered by the user, or executed after a kernel panic; it is faster than usual reboot and can improve availability of the kernel itself, or can be employed by kernel developers to shorten development time. Moreover, the new kernel may differ from the rebooting kernel: it can be an older and well-tested Linux kernel release (useful when a new release is just introduced in a production environment), or can be an instrumented kernel for automatic fault treatment.

At boot time the initial kernel allocates a memory region; then the `kexec_load()` system call is invoked by the user in order to load a second kernel in that region. The loaded kernel can boot from that location without moving, and so the old memory image can be mostly preserved for subsequent analysis when the second kernel is executed. The Kdump patch (also included in the kernel) does such analysis: particular actions are taken by `kexec()` to not overwrite the previous kernel when executing the second kernel. The memory region needed for boot (the first 640 KB for x86 architectures) is saved in a pre-allocated backup region (together with processor registers), and will be available in future.

After booting a customized kernel (*capture kernel*), the original kernel's memory image is made available in ELF (Executable and Linking Format) structure through the `/proc/vmcore` virtual interface, and can be analyzed by standard utilities such as `gdb` and `crash`. The capture kernel can make

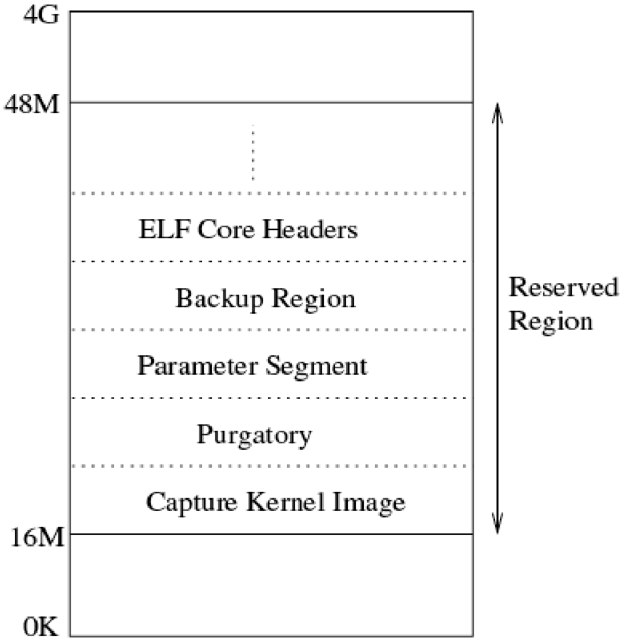


Figure 2.1: *Memory organization for kexec/kdump.*

a dump in different ways: it can store the image on the disk or through a network using user-space programs defined by the system administrator (e.g. a simple shell script can execute the `cp` command on `/proc/vmcore`), similarly to other tools such as `diskdump` and `netdump`. Dumping can also be done before trying to mount a corrupted root filesystem, with utilities in the initial ramdisk. Moreover, the raw memory image can be accessed through the virtual device `/dev/oldmem`.

A matter of interest is the reliability of the dump operation. When the initial kernel goes into a panic, it can not longer be trusted: its state may be corrupted, and execution of critical code such as device drivers can damage hardware and file systems, or interfere with network activity. By running a new bare capture kernel instead of the unstable kernel, the reliability of any dump or recovery action is increased. Moreover, before capture kernel execution, an integrity check is made on the capture kernel's code by the code in the `purgatory` segment (it setups the backup region and the capture kernel, and applies an hash function on this kernel). Execution in a reserved memory region protects the capture kernel from ongoing DMA operations (on x86 architectures, DMA uses addresses in the range between 0 and 16 MB). Nevertheless, fast reboot can be risky because hardware is not correctly shut down, and driver initialization of capture kernel can fail; more work should be done on drivers to guarantee correct initialization after a panic.

2.3 OProfile

A profiler is a tool for applications' performance analysis: it monitors the dynamic execution of a program, measuring the amount of time spent in each section of code. The main objective of a profiler is to identify performance bottlenecks, in order to optimize critical parts of the program code. In general, profilers collect information about events occurred during code execution: in fact, profilers are also used for cache misses and branch mispredictions counting.

Profilers can also be useful for failure analysis, because the code executed in a short period previous to a failure can be correlated with the failure itself. It can also reveal what parts of the kernel were stimulated by a workload, what is the total average system load (e.g. number of tasks in execution, or CPU idle time) and what is the kernel overhead on the overall system load.

OProfile [18] is a system-wide, fine-grained profiler included in the Linux kernel, that can profile both applications and the kernel itself. It is a so called *sampling profiler*: at regular intervals, it collects information about the current task in execution. Instrumentation profilers compose another category of profilers; they modify the original source code in order to collect information (such as procedures execution times). An example of instrumentation profiler is `gprof` [6].

In comparison to sampling profiles, instrumentation profilers are more accurate counters: the instrumentation code is executed exactly once for each event happened. The main drawback is that the instrumentation of code can alter performance measurements: in modern superscalar processors, with out-of-order-execution, any code addition can significantly affect execution, especially for small routines executed frequently (this is the case of a kernel). Sampling profilers can sometimes miss an event, and their approximated measurements should be considered as a statistical result, which is significant if the code is executed for long periods of time. These profilers can be useful thanks to their lower CPU overhead: timer interrupts are executed to store the Program Counter, that can be later correlated with the original source code, and the sampling frequency can be chosen to best accomplish the accuracy-versus-overhead tradeoff; the profiler execution still affects the instruction flow (interrupt requests can flush CPU pipelines), but temporal measurements, if significant, are more realistic for short procedures. The result is a relative measure instead of an absolute one, because it is expressed in terms of ratio between procedure's samples and total samples.

OProfile can be utilized for kernel profiling. If the kernel is built with debug symbols, it can report the number of samples collected for each routine (even for each instruction) in the source code. OProfile supports hardware performance counters: modern CPUs reserve some registers for event counting, that are increased by the hardware when a particular event occurs (e.g. the execution of an instruction, a cache miss, and so on), and a Non-Maskerable Interrupt (NMI) is generated if the count overflows a threshold; the events and the threshold value can be configured by software. NMIs are useful because parts of kernel in which interrupts are disabled can be sampled, too. Interrupt delivery can be slightly delayed, so an event might be accounted to a different near instruction; this and other inaccuracies to take in account when interpreting profiles are described in the OProfile user manual.

An user-space daemon collects profiling data from the kernel, and then dumps it to the filesystem. Profiles also can be organized by profiling session, and separately examined; a periodical dump routine can save profile collected in the period of time since the last dump. Moreover, OProfile has a limited support for call graphing: for each sampled function, it can show the number of times it is called by each other sampled function, and what amount of time is spent by the function in each sub-routine called.

In conclusion, OProfile can be useful for profiling low level code such the kernel and its modules, or libraries, and even if its measurements are approximated, it still can yield representative aggregated information about system load and kernel activity.

2.4 User Mode Linux

User Mode Linux (UML) was defined by its author “a port of Linux to Linux”. It is a modified Linux kernel that can run as a process in a Linux environment (*host*), and it appears as a virtual machine to its processes: they cannot distinguish it from a physical machine, and they are executed unmodified.

UML [20] is similar to other industrial solutions such as Xen [23] and VMWare [22]: it enforces isolation and resources abstraction to programs running on the same physical machine. UML can only run Linux on a Linux host: so its range of applications is restricted, but at the same time it can exploit facilities of the underlying host and UML kernels to achieve better performances. Its main employment is in kernel development: because an UML kernel run as a common unprivileged process, it can be modified and tested without worrying about damages in software and hardware resources, and so can be utilized for development of features not related to physical devices, such as filesystems, schedulers and memory management; moreover, its virtual networking support may be a substitute to the employment of physical network equipment. As its commercial counterparts, several UMLs can be utilized to offer hosting to customers’ applications: each service can be executed as in a private environment, without interference from other processes, and providers can consolidate hosting on few physical machines, exploiting performance of modern multiprocessor servers and lowering setup and management costs.

UML can also be useful in the context of dependability. Resource virtualization can limit error propagation between services on the same machine, as it would be if they executed on different physical machines (virtual and physical machines and their services can only interact through the network); isolated services’ execution (namely, jails) can be used to prevent malicious users, which compromised a virtual machine, from accessing resources on the host system; the host can apply automatic recovery countermeasures even if the guest system failed, and collect data about failures; fault handling and recovery of distributed systems can be tested within virtual machines’ networks before physically deploy them. However, UML is not suitable for testing and stressing of real device drivers, because an UML kernel works with virtual peripherals.

An UML kernel can be created by compiling default Linux kernel source code for an artificial target architecture (using the `ARCH=um` make parameter). The result is a standard ELF executable, that can be run on any vanilla Linux system. The UML kernel attaches a console to the standard input and output of the host process running it (i.e. the xterm in which the UML binary is executed), and other virtual devices of the UML kernel can be attached to host system ones (such as consoles and pseudo-terminals). Each virtual block device (such a disk) on the UML kernel is backed up on the host system by a common file, and reads and writes are redirected to host file's blocks. UML user space utilities can make backup of virtual disks' contents, increase their size, and configure them for block sharing between virtual machines, namely the Copy-On-Write mechanism. The UML kernel access these blocks, formatting them with any supported filesystem, without affect the underlying host filesystem: the UML filesystem is seen by the host as a simple file. The UML kernel make use of `O_DIRECT I/O` [BC05], in order to bypass host kernel caching (if not so, there will be useless cache entries duplication because UML kernel implements its own page cache). At least one virtual disk has to be configured as the root filesystem in order to boot an UML kernel, and must include all libraries and system files needed to run a Linux operating system. In general, the UML kernel is built only with a small group of drivers, to access virtual devices exposed by the UML kernel itself (and indirectly mapped to host resources).

The UML kernel and its processes are executed as native processes of the Linux host system; UML processes' system calls to the host kernel are redirected to the UML kernel using `ptrace()` [32] [33], and executed in that context. There are two main approaches by which the UML kernel can control UML processes execution; a third mechanism, called Tracing Thread, is deprecated. The first execution mode is indicated as `skas3` (the acronym stands for Separate Kernel Address Space). The UML kernel code and state resides in a separate host process, and all UML processes are executed in the context of a single host process.

Unlikely to an usual Linux kernel, the UML kernel address space is not mapped in the higher 1 GB of the (user) address space of each UML process; if it were so, the UML process program would have the opportunity of read and write the UML kernel memory image, because the host process executing that program would access the whole user address space of the host process (containing both UML process' and UML kernel's address spaces). For example, in 32-bit systems, where the total address space size is $2^{32} = 4$GB, the Linux kernel is typically configured to use the top GB addresses (in the address interval $[2^{24}, 2^{32} - 1]$) as kernel address space, and the remaining 3 GB address interval ($[0, 2^{24} - 1]$) as user address space; in the scenario previously depicted, both the UML kernel and an UML process are allocated in the host process user address space $[0, 2^{24} - 1]$, and the host process that executes the UML process can read and write the whole address interval, which is a security threat for the UML kernel. Moreover, the limited 3 GB address space available (this is the case of Linux on x86 systems) is a strong constraint for UML kernel and processes, and require use of High Memory to manage a large "physical" primary memory, with a performance drawback (to be more precise, UML kernel's primary memory is not a real physical memory, but a portion of the virtual memory allocated to the host process wherein UML is executed). Instead, the UML kernel address space is allocated in the (wider) user address space of a distinct host process, inaccessible by a different host process that executes the UML process' program.

Separating the address spaces requires a particular patch to be applied to the host kernel in order to make the UML kernel able to do some operations on a host process containing an UML process, such as page fault handling. The `ptrace()` system call is modified, so the UML kernel, passing a specific flag, can receive `SIGSEGV` signals caused by an external UML process; moreover, the UML kernel can modify the memory mappings for an UML process (changing the address space descriptor of the task), and retrieve swapped pages from a virtual block device. In general, the UML kernel can create (and then delete) an address space without allocating a new process on the host system, and can substitute the address space descriptor for the child process: so it can manage several UML processes with only one host process, lowering the overhead for the host kernel (e.g. a context switch of the UML kernel does not require a full context switch on the host kernel).

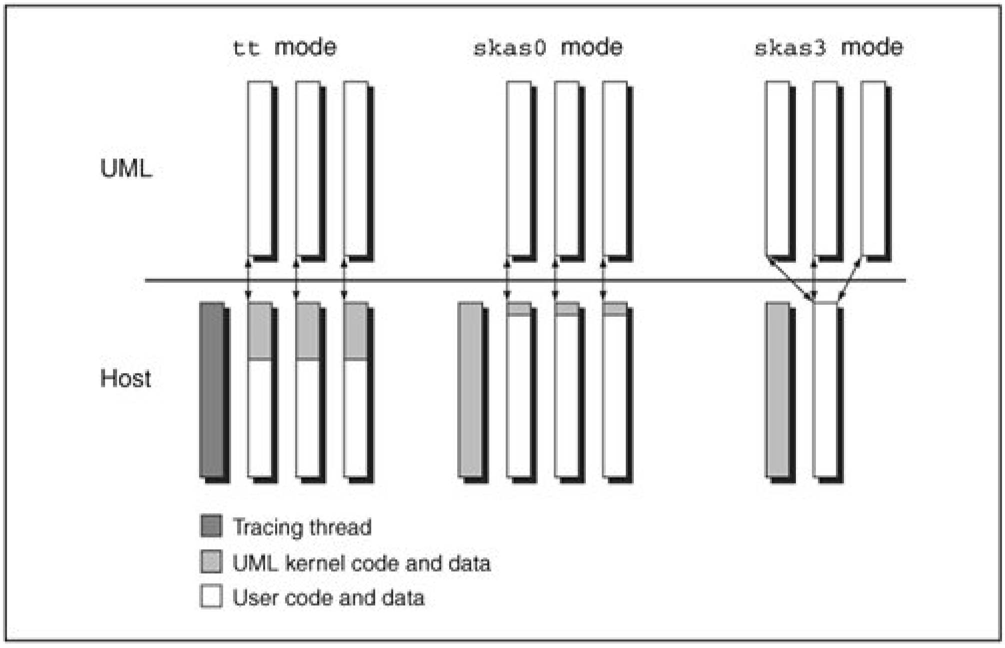


Figure 2.2: Address space layouts for different UML execution modes.

Another execution mode is called `skas0`; it realizes a mechanism similar to `skas3`, without patching the host kernel. Again, UML kernel is executed in a separate host process, but each UML process is executed in its own host process. Each UML process address space include some code (in the top two pages) to allow address space updates; the UML processes cooperate with the UML kernel, and when a `SIGSEGV` is handled by the code injected in the UML process, the UML kernel is notified and performs the adjustments to the process memory image needed for page substitution. In fact, the primary memory of the virtual machine is `mmap`-ed to a file in the `/tmp` directory of the host system; this file is accessed by the UML kernel for page replacement, and therefore updates are seen by UML processes; then the UML process makes the needed virtual memory mappings through `mmap` on the same file, as requested by the UML kernel. To improve performances, the `mmap`-ed file should be stored in a ramdisk (e.g. tmpfs) to keep it in physical memory instead of the disk. This approach require that each UML process is execute inside a distinct host process, because the UML kernel can not replace the address space descriptor of processes; in general the `skas0` approach is less efficient, and skas3 is suggested to optimize system resources utilization.

In a UML system, time flowing is virtual. The UML kernel and its processes are executed within host processes, running in a time sharing host system. Time clock for the virtual machine is increased only when UML is scheduled by the host kernel, therefore the length of a virtual time period differs from the real time elapsed. This can be tolerable for applications that do not need to interact with processes on other systems (virtual or physical), otherwise synchronization is needed. Temporal offset can be compensated if the virtual clock is synchronized with the host clock at each schedule time slice: the UML kernel does so by invoking its timer interrupt routine for each past clock tick since the latest time update, and the time is increased in a non-uniform mode; this behavior can be enabled at UML compile time. The virtual time flowing can be useful for debugging: when data is dumped from an UML kernel (e.g. using printk()s or gdb), the time in the UML machine is halted, and it is restarted after debug operations by the host, so UML execution is not delayed; this property avoids timing bugs (such as race conditions) not to be activated because of delays introduces by the debugger.

The UML binary which contains the UML kernel can be executed through gdb for debugging purposes; as any other user space process, gdb is allowed to set breakpoints on it, inspect the program variables, and so on. Tracepoints are an useful feature for field data collection: internal data can be automatically dumped when a point in the program is hit by the execution flow, similarly to breakpoints; moreover, gdb can do debug from a remote machine through a network connection. Signals used internally by UML, such as SIGSEGV and SIGUSR1, should be ignored by gdb. The UML management console, an user space tool, can also be used to get runtime information on UML kernel, to hot-plug virtual peripherals, to modify UML configuration, and to receive notifications from an UML kernel.

UML supports several mechanisms for networking: an UML virtual machine can be configured in order to exchange packets with the host machine, with other UML virtual machines on that host, and with other physical machines connected through a real network. Different transports are used for packet transmission to and from the host machine: the most efficient and commonly used is the TUN/TAP; the host can then forward a packet to the physical network, or to a different virtual machine. Another popular transport, used for virtual networking between virtual machines, is the switch daemon, an user space process that communicates with all virtual machines through UNIX sockets.

The TUN/TAP transport is an user space interface that enables a process to send and receive packets to and from a network interface (e.g. tap0), by reading and writing on a virtual device file (e.g. /dev/net/tun). Therefore, the UML kernel can open the device file and, through a virtual network driver, can enable a virtual network interface for UML processes; packets sent on the virtual network interface (e.g. eth0) on UML machine will arrive to the host TUN/TAP network interface (tap0), and vice versa. The TUN/TAP transport resembles a point-to-point Ethernet link between UML and the real machine.

In order to connect an UML virtual machine to the rest of a physical network, two configurations on the host machine are suitable: routing and bridging. In the routing configuration, the host machine's routing table is configured to forward packets (originated by and directed to the virtual machine) to and from the tap0 interface, similarly to a physical interface. If the virtual machine has to appear as a physical machine connected to the physical shared media, some workarounds are needed for broadcast-based protocols (e.g. ARP, DHCP), such as user space proxies for application level broadcasts' forwarding. Routing configuration is shown in figure 2.3.

Another solution is the bridging configuration: the host Linux kernel is configured to work as bridge, in order to forward any layer-2 packet incoming on an interface to destination interface(s). The eth0 and tap0 interfaces are attached to the virtual bridge using the `brctl` user space tool, and the host kernel has to be compiled with the Ethernet Bridging support. This configuration is somewhat more sophisticated, but it avoids the recourse to user space proxies and routing on the host machine; moreover, it does minimal computing on packets exchanged by the virtual machine. Bridging is shown in figure 2.4.

UML is a quick and adaptable tool for setting up virtual machines and virtual networks; therefore its applications are almost unlimited. Utilities, like Netkit [17] and other UML add-ons, are available to easily build virtual

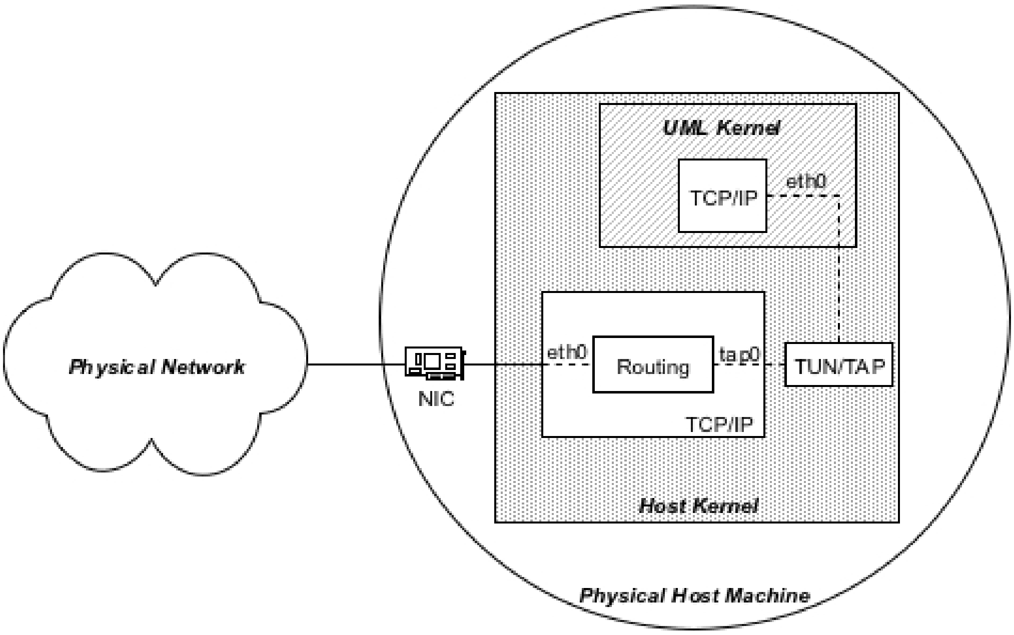


Figure 2.3: *UML networking: the UML virtual machine is connected to the host machine through a virtual point-to-point Ethernet link, and it is routed to the external network.*

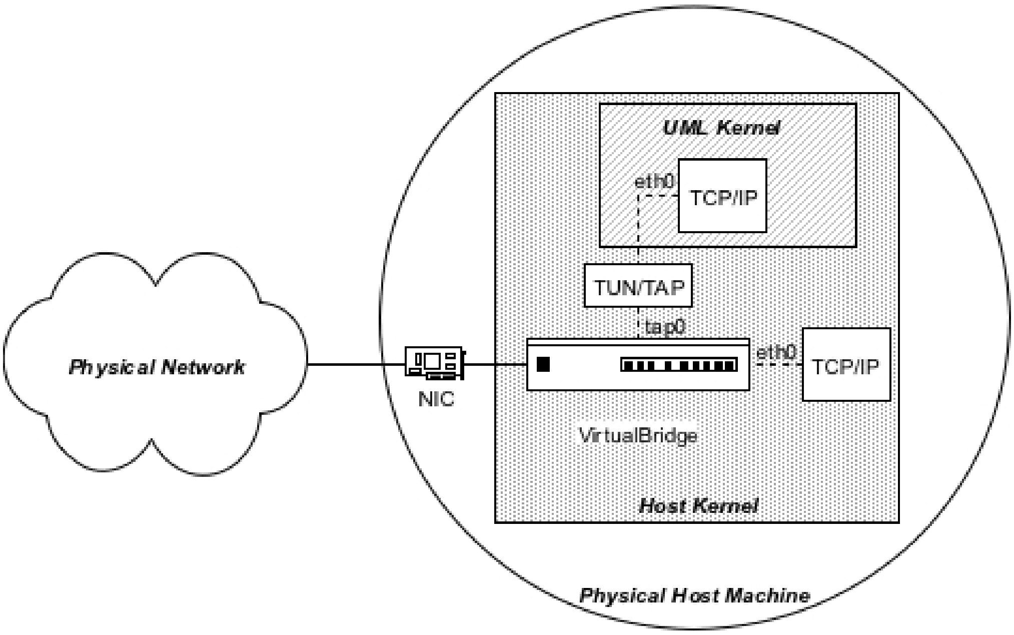


Figure 2.4: *UML networking: both the UML virtual machine and the host machine are connected to a virtual bridge, and share the same broadcast domain.*

distributed systems.

2.5 KProbes and SystemTap

KProbes is a framework, included in the Linux kernel 2.6 series, for tracing kernel execution using dynamic probes: they are debug points defined by a code instruction address, and user-defined handling functions are invoked when kernel execution hits that addresses. Handlers can be used for automatic collection of kernel data, without user intervention and preventing kernel stall; afterwards, brief reports and statistics can be produced for performance or behavioral analysis. The generality of this approach enables KProbes to be applied on any part of the kernel. Dynamic probes are defined at runtime, with neither code instrumentation nor kernel rebuilding, so they can be utilized for field data collection on production systems: they has been successfully tested for large, complex systems such as Symmetric Multi-Processors (SMPs).

Figure 2.5 shows the three parts of KProbes framework. The lowest layer provides facilities that are architecture dependent such as code breakpoints injection, process stepping, handlers invocation, and so on. At the time of writing (kernel version 2.6.20) KProbes is available for the following architectures: i386, x86_64, ppc, ppc64, ia64, sparc64. The middle layer, named KProbes Manager, is architecture independent, and provides an API for handlers' setup and management. The top layer consists of the user defined probe handlers: by loading a kernel mode, an user can add or remove at runtime dynamic probepoints, and read or modify kernel data.

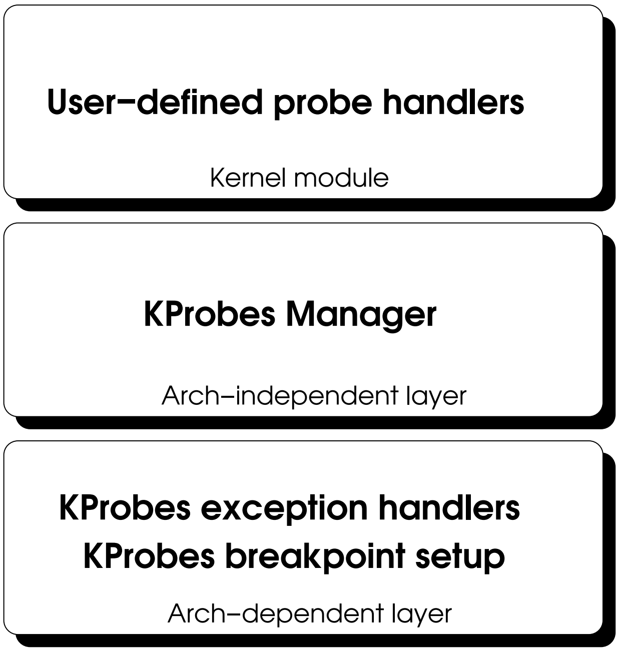


Figure 2.5: *KProbes layered architecture.*

When defining a probepoint, the user allocates a `kprobe` struct, wherein he specifies the probepoint's address and the handlers to invoke. Three types of handlers are available: the first two, pre-handlers and post-handlers, are called before and after the addressed instruction execution, respectively; the third type, fault handlers, are invoked before post-handlers if the probed instruction execution faults (e.g. a page fault occurs): then the fault handler can process the fault in place of the kernel. The kernel module written by the user can register the dynamic probe using the KProbes Manager interface with the kprobe structure just defined.

When a probe is registered, the probed instruction is substituted by an architecture dependent instruction that triggers the KProbes code in the kernel (in the x86 architecture, the INT3 instruction can be used to trigger a trap, likewise a debugger would do). When the execution hits the probepoint, handlers are invoked, and then the probepoint is restored. Dynamic probe execution is summarized in figure 2.6.

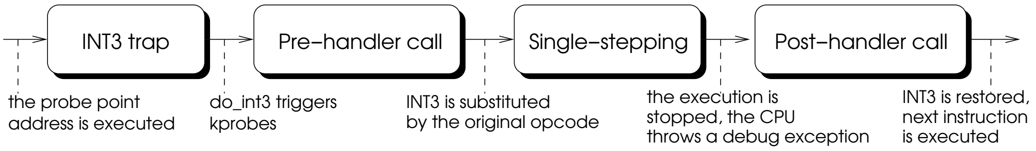

INT3 trap
Pre–handler call
Single–stepping
Post–handler call
the probe point address is executed
do_int3 triggers kprobes
INT3 is substituted by the original opcode
the execution is stopped, the CPU throws a debug exception
INT3 is restored, next instruction is executed

Figure 2.6: *Execution of a dynamic probe for the x86 architecture.*

JProbes are another type of dynamic probes: they are based on KProbes, but JProbes simplify instrumentation of a function entry point. When using a jprobe, the user specifies a probing function with the same signature of the probed function: when the probed function is invoked, the probepoint is triggered, and the probing function is executed; from there, the user code can easily access the input parameters passed to the original function, without doing weird address arithmetic. The original stack frame is then restored and the original function is executed. Jprobes can be registered similarly to KProbes through the KProbes Manager.

SystemTap [19] is an high-level language, simple and loosely typed, for defining dynamic probes: with SystemTap, an user can write a script without the burden to write, compile and load a C kernel module. A SystemTap script defines actions to execute for each probepoint introduced; that code is translated in C, plugged in a kernel module, and automatically compiled and loaded by the SystemTap stap user space tool. The flow diagram in figure 2.7 shows the execution of a script.

When defining a probepoint, the user can explicit the instruction address to probe like in KProbes, or it can use symbols (e.g. the name of a system call) defined in SystemTap libraries (namely *tapsets*) or in kernel debug-info objects. Such probepoints are called dwarves: an handler is executed when a function entry or exit point, or a source code line in the kernel or

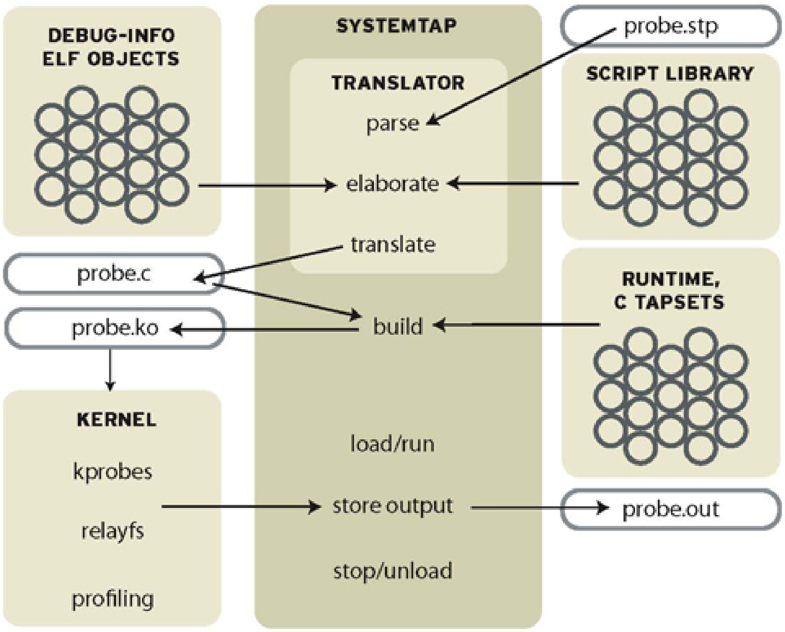


Figure 2.7: *Phases of a SystemTap script execution.*

in its modules, are encountered. Symbols simplify probe definitions, because the user does not directly work with memory addresses; moreover, scripts can easily be ported to other kernel versions or architectures, and reused if the current kernel is slightly modified and recompiled. Another type of probepoints are `timers`, that execute an handler in an exact point in time or periodically, and `begin` and `end` probes, that are triggered at module loading and unloading. Information produced by the handlers in the kernel are gathered by the `stpd` user space daemon through `relayfs`, a virtual filesystem (also included into version 2.6 kernels) for simple and efficient access to kernel buffers.

The LKET (Linux Kernel Event Trace) tool is based on SystemTap, and it is included in SystemTap tapset library. It defines a set of standard probepoints (*hooks*) in pre-defined kernel places, to collect information about the execution in the form of a *trace* file that is later examined by the user. The trace is saved by lightweight KProbes' handlers in a compact binary file, thus increasing the efficiency of user-space data collection. Several hooks are defined for monitoring of:

- System calls;
- Process creation;
- UNIX signals;
- Page faults;
- I/O scheduling activities;
- Task scheduling activities;
- SCSI and network devices activities;
- NFS and RPC activities.

The user can specify the events of his interest through a SystemTap script, and customize hooks and trace data by defining further probes.

KProbes are a robust mean for inspecting kernel execution. However, ensuring robustness can be a bottleneck for kernel performance, so KProbes are not the best suited tool for kernel profiling. In fact, when a probe is triggered, process preemption and interrupts are disabled, to make the probed instruction execution atomic without expensive state saving and restoring, thus increasing interrupt latency and lowering system throughput. Moreover, handler execution on SMP and multi-core systems is serialized (the internal probepoints' hash table is locked, avoiding processors' mutual interference on execution of instrumented instructions), and two synchronous exceptions are generated for each probe handling, increasing the delay and the throughput loss due to KProbes.

2.6 Other tools and related work

Articles [36] and [31] give an in depth view of kdump organization; other introductory articles on kexec and kdump are [27] and [28]. Another project similar to kdump is Mini Kernel Dump [14]; other projects for kernel dumping are [10], [16] and [4], but they are considered less reliable because the dumping is executed by the failing kernel itself. Dumps can be analyzed with `gdb` [5], or with the specialized `crash` utility [2].

A general introduction to kernel and application profiling and coverage tests on Linux system can be found on [Bes05]; older debugging tools are also referenced in [24]. OProfile is explained in [34] and [Bes05], and the official web site [18] contains some examples of use; a deeper discussion on OProfile utilization and limitations can be found on the official documentation on [18].

Many resources on User Mode Linux, such as tutorials, kernel builds and filesystem images can be found on the site [20]. The book [Dik06], by UML's author, gives an exhaustive discussion on design choices, configuration scenarios with examples, internals and future development plans. Two howtos on the use of `ptrace()` system call for process tracing are [32] and [33]. Netkit is a didactic project based on UML for virtual networks simulation; it is available at [17].

Introductive articles on KProbes architecture and utilization are [30], [25] and [35], with example kernel modules. The SystemTap tools can be downloaded from [19]; see [26] for an howto on SystemTap scripting. An ongoing project for Linux kernel tracing is LTTng [13]: the kernel is patched to generate events with low overhead through static source-based probes, such that high-level analysis on kernel execution (schedule decisions, process switches, I/O, . . .) can be done later on the event trace. Another project for kernel tracing by static probes is LKST [11]. Finally, the Djprobe project [3] provides a mechanism for dynamic probing of the kernel similar to KProbes, but the `JMP` instruction is employed to instrument the executable instead of `INT3`, and thus introducing a lower overhead.

2.7 Conclusions

In this chapter several technologies for kernel instrumentation and analysis have been described; these differ in terms of performance, complexity, range of applications, hardware support, reliability, accuracy of results, so the choice between them depends on analysis objectives. However, these tools can be successfully employed for both kernel development and data collection from real and virtual systems. Their strengths (+) and drawbacks

(−) are summarized in the following:

1. Kexec and Kdump

   + Automated and fast intervention mechanism after a failure;
   + Reliable under critical software failure conditions;
   + Complete information about execution context after the failure;
   − Post-mortem only analysis;
   − Memory pre-allocated to the capture kernel is wasted;
   − Not reliable if the hardware is faulty.

2. OProfile

   + Low overhead, hardware-based profiling;
   + Useful for kernel profiling and workload analysis;
   + No software instrumentation;
   − Statistical, approximated results;
   − Little information about the kernel state is collected.

3. User Mode Linux

   + Runs a Linux system in a virtual process-transparent environment;
   + Complete kernel debugging;
   + Protection of hardware and software resources;
   + Enables virtual network testbeds;
   − System call overhead, memory consumption;
   − Careful hardware dimensioning and software configuration needed;
   − Hardware drivers' code is not taken in account.

4. KProbes and SystemTap

   + Dynamic on-line kernel analysis;
   + Complete information about the execution context;

- + Simple and powerful probepoints and handlers definition;
- − Not-negligible kernel execution overhead;
- − Not suited for performance analysis.

CHAPTER 3

Evaluation of failure detection capabilities of Operating Systems

You cannot have a science without measurement.

RICHARD W. HAMMING

3.1 Introduction

In this chapter, we discuss about the facilities commonly provided by operating systems to deliver information about the state of the system to users and applications, which we refer to as logs. Several works on dependability analysis (FFDA) highlighted that data recorded in system logs is often incomplete or imperfect, and the related dependability measures can be potentially misleading [SK05] [BS95]; moreover, in the majority of cases they do not allow to go back to the causes of failures [XKI99]. Failure detection is also a prerequisite for proactive mechanisms to improve system dependability, such as these described in [HBG$^+$07]. Our goal is to evaluate the actual effectiveness of existing detection mechanisms provided by the system (e.g. `printk()`s by the kernel, and logs of daemon processes). Then, we try to identify symptoms in the system behavior that could be exploited to improve the detection of errors in the system. Fault injection is used to compare the effectiveness of different detection mechanisms. We consider two criteria to characterize a failure detector, derived from [CTA02]:

Coverage the conditional probability of detecting a failure when a failure exists;

Latency the time required to detect the occurrence of a failure.

Moreover, the failure detector should support fault isolation, by which the root causes of a failure are determined: an error in a software component can cause errors in other software components (e.g. a misbehaving component could corrupt data shared among multiple components), and the error can propagate again before the system manifests a deviation from the correct service specification, or a warning is produced by detection mechanisms, in a software component unrelated to the original fault. Therefore, a detector should provide useful hints to go back to the offending component, and a low detection latency is essential not to lose information about the location of the first error (if several errors are present, the diagnosis can be mislead), and so to apply fault treatment techniques and to provide corrective maintenance.

We consider the operating system (OS) as an aggregate of different sub-components: a fault can be located within a sub-component, and its activation (the occurrence of an error) can lead to a wrong behavior experienced by other sub-components (the occurrence of a failure of the faulty sub-component); the failure represents a fault for other sub-components, which can lead to other errors and failures. When a fault affects the overall operating system's behavior, it is causing a *system-level failure*, i.e. a failure at the interfaces between the OS and other external components and the users. A failure of a OS sub-component can also be seen as an error state internal to the OS itself. Both coverage and latency can refer to failures occurring in a single sub-component, or to system-level failures (see 3.3.1). In the context of operating systems, we account as software components the device drivers and major kernel subsystems (e.g. memory management, CPU scheduler, filesystems) defined by the general architecture of the system.

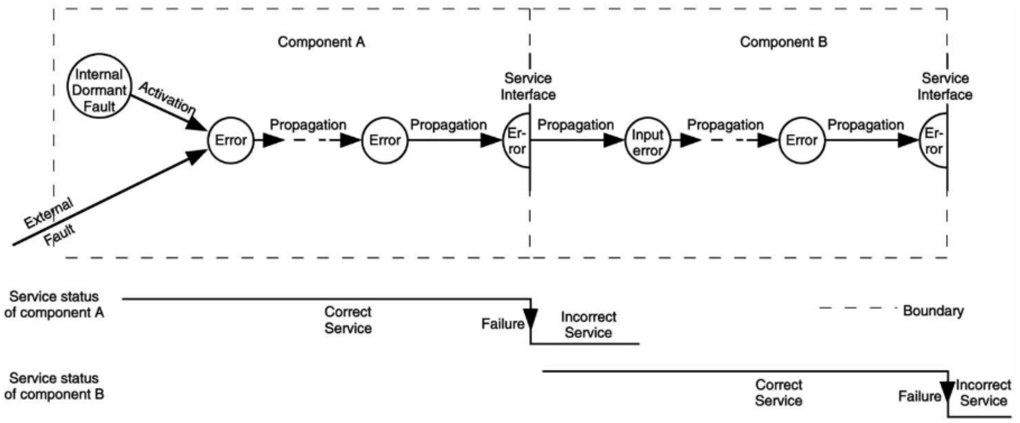


Figure 3.1: *The dependability chain illustrating the relationship between faults, errors and failures in a component-based system [LRAL04].*

3.2 Fault Injection Techniques

3.2.1 An overview of fault and error models

In order to evaluate the ability of the operating system to report failures, we execute fault injection experiments to speed up the activation of latent faults by simulating the effects of defects in the kernel code. Different approaches are possible when injecting faults on a running system; we distinguish between them by:

- the type of the injected fault;
- the methodology by which a fault is injected.

Moreover, the fault injection experiments can be further classified by the system's life stage in which they are executed: *system design*, *prototipal* and *operational* phases; since we focus on commodity (COTS) operating systems, we will consider fault injection on the final system (the one executed on the field).

In computer systems, both hardware faults and software faults could be injected (the latter are defects in the source code that cause an error state when executed under particular circumstances, namely defect triggers); in this work, we aim to study the effects of software defects, so we only consider injection of them.

Several methodologies are available for the injection. Hardware faults are deeply understood, and the most frequent hardware error causes are known and well modeled; they can be physically injected (e.g. the use of a radiation source to induce single event upsets on integrated circuits), or their effect can be emulated by software (e.g. by bit-flipping a data word through the execution of a program). Instead, software fault injection is a much more difficult matter, because of the great variety and complexity of error causes and triggering conditions in computer programs.

The most popular and simple technique to emulate software faults is the direct injection of the errors caused by a software defect. When injecting errors, the state of the executing program is artificially altered in order to mimic the effects of the activation of a software fault; the state modification may concern the data and the instructions handled by the program (because the most part of programs are not self-modifying, the state manipulation is usually made on the data). The injection of errors in the data can be performed on the contents of the memory (e.g. variables, constants, stack), or it is performed on the parameters of procedure invocations. Common error models (compared in [JSM07]) for error injection are:

Bit-flips one of the bits representing a value is inverted (e.g. the first of 32 bit representing an integer is modified from 0 to 1);

Data type errors a variable is assigned a “faulty” value in a well-chosen subset of the values defined by its type (e.g. an integer variable is assigned to 0);

Parameter fuzzing a variable is assigned to a random value in the set of values defined by its type (e.g. an integer variable is assigned a pseudo-random integer value).

In general, in order to obtain an injection representative of real faults, the memory alteration should be done on a high abstraction level, i.e. we can modify a data word if it contains the value assigned to a variable defined in the source code of a C++ program, but we do not alter the memory by randomly choosing a memory location.

These fault injection techniques are often employed for testing of components by parameter alteration at the interfaces (e.g. robustness testing, see figure 1.5), because the reuse of components in COTS-based design exposes software faults not yet discovered, due to the new environmental conditions and components interactions [MP05] [MDB$^+$07]. In a COTS operating system, the interface to other software components is defined by the SCI (System Call Interface), by which its services are invoked. The robustness of the operating system to exceptional inputs can be evaluated by alteration of invocation parameters of system calls by user-space processes, or by ad-hoc automated tests that try several different combinations of input values (similarly to black-box testing). In [JAC$^+$02], experiments using bit-flips and data type error injection at SCI were made; comparable results were produced in terms of execution outcomes (e.g. percentage of crashes, or hangs), although bit-flips required a much larger number of tests than data type, and therefore a greater amount of time is needed for the campaign. Nevertheless, the robustness testing at the SCI is unlikely to discover short-term failures in mature software: our preliminary experiments on recent versions of the Linux 2.6 kernel family with the Ballista robustness testing suite did not discover failures among the tested subset of invocations. Moreover, as observed in [JAC$^+$02], externally injected faults at the SCI are not able to emulate (i.e. to cause similar effects) internal faults, like faults in device drivers.

Error injection in function parameters was also applied in several work to emulate internal faults, by targeting the interface between the main kernel subsystems and pluggable code like device drivers. Compared to SCI-level, the drivers' level (namely DPI, Driver Programming Interface) is more prone to robustness failures, because of implicit assumptions by the kernel on the drivers' behavior: to improve performances, most systems avoid the costs of extensive parameter checking. In [JSM07], different techniques (bit-flips, data type errors, parameter fuzzing) are evaluated by implementation complexity, execution time, and observed measurements in terms of failures' impact and error propagation (the degree to which an error spreads other errors in the system, namely *diffusion*); it is shown that bit flips are easier to implement and produce the major number of failures covering more operating system's services than data type errors and fuzzing, but it has also the highest execution time, and the authors point out that there exist robustness faults discovered by tests not generated by bit-flips; in order to reduce the execution time while maintaining similar percentages of observed failures and measured diffusion, an error model is proposed that mixes bit-flips and fuzzing. Data type produced the minor number of failures, and it requires a preliminary analysis of interface's data types in order to establish the subset of tested values for each type (e.g. boundary values such as 0, 1, 0x$FFFFFFFF$ for signed 32-bit integers). Fuzzing produces results about diffusion similar to bit-flip, with a lower number of discovered "breakable" services; it is argumented that random values are far off legal values and then easier to be found by internal checks, but they can find robustness failures other than the ones of bit-flips and data type errors; moreover, fuzzing also requires preliminary work to establish the number of injection cases for each service (by evaluating the stability of the results when increasing the number of tests), and the repeatability and the uncertainty of the tests should be quantified in order to compare the results with different studies [BCFV07]. Instead, bit-flips and data type errors do not add further uncertainty to results (see below for a discussion on statistical interpretation of the experiments).

Another approach for software fault injection experiments is based on mutations of executable code to reflect common mistakes in high level source code. In [DM06], an analysis of a collection of real software faults on several programs is conducted to define the constructs (*operators*) most representative of frequent software defects; then the G-SWFIT (Generic Software Fault Injection Technique) is introduced, which allows the identification of locations where a given fault could exist, searching for known patterns in the executable binary code, and the following injection of defects by mutating the low-level instructions in the executable (figure 3.2), based on the statistical distribution of real faults (figure 3.3).

Differently from error injection, this approach guarantees that errors introduced in the system are caused by actual software defects in the code, and that injected defect types are representative of common errors occurring in programs. However, injected defects are permanent faults and not transient

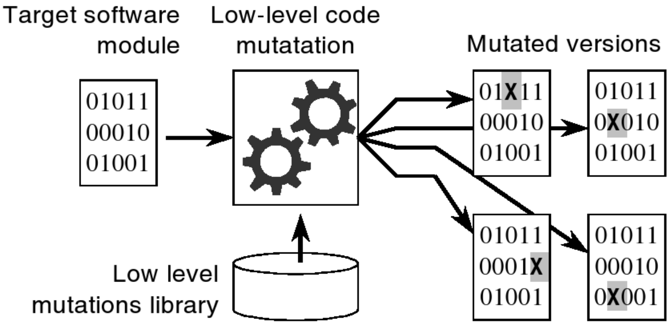
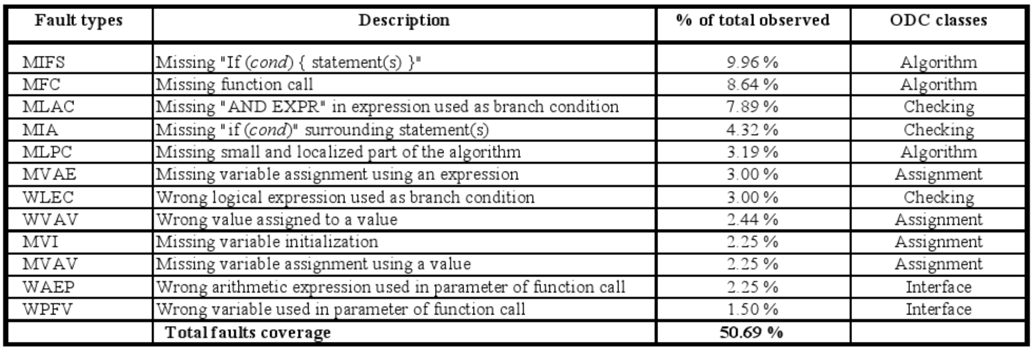


__Figure 3.2:__ Software fault injection through the G-SWFIT (Generic Software Fault Injection Technique).



Fault types	Description	% of total observed	ODC classes
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

__Figure 3.3:__ Common types of high-level software faults found in several programs [DM06].

ones, because they consist of the removal/addition/alteration of operators in the binary program code: they last for the whole period of an experiment, and they are triggered each time the fault location is executed; instead, transient faults represent a major source of failures, and this injection technique does not take care of reproduce triggers that activate these faults; so, the applicability of this approach is subordinate to the type of defects the experiment is expected to reproduce. Mutations were applied in dependability benchmarking [MDB$^+$07] and COTS risk assessment [DM04], in which the impact of software faults on operating systems is evaluated.

In [CC96], a general approach to error injection is described, in order to accelerate fault injection experiments while reflecting the field data about real faults. The data provides:

- a classification of all reported (and fixed) defects by type (the ODC categories *Checking*, *Assignment*, *Algorithm*, *Timing/Serialization*, *Interface*, *Function*) and consequent error type (i.e. *single address*, *single non-address*, *multiple*, *control error*, see table 1.4);
- the component (and its internal module) in which each defect was found;
- the trigger for each error type (*normal mode*, *startup/restart*, *workload-/stress*, *recovery/exception*, *hw/sw configuration*).

The methodology indicates *where* and *when* inject an error representing *what* fault type (e.g. tables 1.1, 1.2); it consists of the following steps:

- first, the joint probability distribution between faults and errors is estimated (by calculating the relative percentage of each pair fault/error);
- then, for each component and for each module, a list of potential fault injection locations is obtained (by an automatic parsing tool that classifies each statement of the code to an ODC defect type category and an error category);
- a subset of fault locations is randomly chosen, based on the distribution of field faults in the components and modules, and the joint distribution of fault/error pairs;

- an operational profile (a statistical description of system usage) is defined: it reflects the utilization of each system's service in each triggering context, for example by weighing the relative occurrence of each trigger.

In order to correctly emulate software faults (thus excluding hardware faults), the injection of an error is synchronized to the execution of the code in the associated fault location; the error represents the activation of a (supposed) fault type in the selected location. In order to shorten execution time, software engineering techniques are employed to provide inputs to quickly execute the code in the fault location (it is similar to the problem of white-box test coverage), then the error is injected (the injecting procedure is activated by software or hardware traps) and a workload representing the operational profile is executed.

The described approach for error injection is general, and it enforces the representativeness of real software faults (both defect types and trigger are considered, and hardware faults are factored out) in experiments. Moreover, the direct injection of errors decreases the duration of the experiment because it avoids to wait for fault activation (the transition 1 in figure 3.4). Unfortunately, its applicability is limited by the availability of representative field data (it is an essential requirement for whatever injection technique that should be representative), and so in this work it only constitutes an ideal model to which compare different and more feasible approaches.

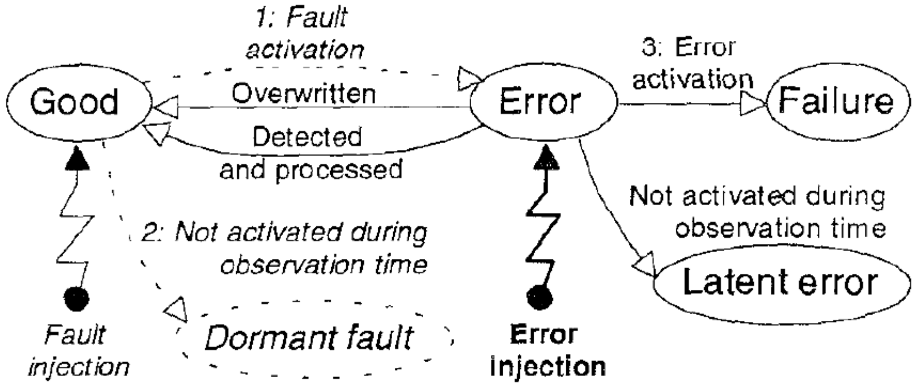


Figure 3.4: Evolution of a system subject to the injection of a fault or an error.

3.2.2 Choice of a model

We now examine the different fault injection techniques described previously, to select the one that best suits the objective of this campaign, i.e. to evaluate the ability of the operating system to detect failures due to software defects present in the operating system itself.

First, we make a clear separation between the component in the System Under Benchmark (SUB), which is the target of the injection, and the component under evaluation (the Benchmark Target, BT); in order to avoid the problem of evaluating an altered component (e.g. we do not want to modify its executable code), we inject a fault in a different one, that will propagate errors to the component being evaluated. Thus, we have to determine in the operating system a target component of minimal size (to leave the most of the operating system unaltered by the injection of a fault) and well defined (the boundaries of the component are explicitly specified).

The device drivers are chosen as Fault Injection Target (FIT). They are an optimal (and very popular) target for such experiments because:

- they are a well defined and distinct component of the operating system;
- they are a big part of the kernel source code, and a major source of failures;
- the operating system interact very often with them, and so they grant an high fault activation rate.

Several works dealt with the impact of failures in device drivers on dependability of operating systems. Indeed, a study on software defects in the Linux operating system [CYC$^+$01] showed that

- device drivers account for the majority of the code size, and have a defect rate up to three to seven times higher than the rest of the kernel;
- defects tend to cluster (i.e. several defects occur on the same software module);
- recently-added files have a defect rate twice than the older ones (i.e. introduced in past kernel versions).

Therefore, it is expected that device drivers will continue to represent a major source of failures, because they are constantly added and updated due to the typical short life cycle of hardware components.

In the Minix operating system (§ 1.5.2), failure resilience is achieved by isolating device drivers from other system components: they are executed within a private address spaces protected by the MMU (Memory Management Unit of the processor), thus preventing memory corruption due to driver misbehavior; they have minimal privileges for accessing to physical (e.g. DMA, Direct Memory Access) and virtual (e.g. IPC, Inter Process Communication) resources. Moreover, driver encapsulation enables for recovery procedures that replace a failed driver process by rebooting it [HBG+07]; nevertheless, failure detection is a prerequisite for such mechanism, because recovery procedures can be applied only when the occurrence of a failure is noticed by the system, and information is available about the failed component.

Other works also recognized device drivers as critical components, and considered them as a fault injection target for operating systems evaluation: in [DM02], the G-SWFIT technique is applied to observe the relative occurrence of different failure modes in three operating systems (the BTs), by injecting faults in device drivers (the FITs). In several publications [AAF04] [KJA+04] [MN07] [JS05], robustness testing is applied at the interface between device drivers and the kernel to evaluate the ability of the latter to handle bad inputs, stressful conditions and faulty drivers.

In this work, we adopt robustness testing as a mean to emulate activation of software faults in device drivers; motivations that encourage this approach instead of mutations (i.e. G-SWFIT) are:

- The error injected by a robustness test is transient; after that a parameter or return value is corrupted, no further error injection is made, thus leaving the system free to execute and potentially fail. This behavior emulates a transient fault, which is unexpectedly triggered by complex causes and then disappears. When doing mutations, the binary code is persistently modified, therefore the fault is triggered each time the mutated code is executed, from the beginning to the end of an experiment.

- Robustness testing enables a more thorough and efficient testing of the possible interactions between the drivers and the kernel. While mutations are representative of common defects in the source code, it is not guaranteed that there will be an error propagation to the BT (i.e. there will be a failure of the OS sub-component), neither that they will lead to errors representative of defects due to the integration of drivers in the system (e.g. implicit assumptions on the DPI or on the hardware) if they are not supported by field data.

- In robustness testing, the trigger that activates the error injection can be either a temporal or a spatial trigger, and more complex triggering conditions can be defined.

- The injection of an error by parameter corruption allows to measure precisely the time in which the error is propagated to the benchmark target, i.e. the time in which the driver fails. Thus we can measure the time after that the presence of the error (or a failure caused by it) is logged by the system.

- The FIT is left unchanged, so it can better reproduce further interactions between the kernel and device drivers after that an error is injected.

- Error injection experiments are easy to setup and to understand.

Nevertheless, there are also disadvantages that limit the applicability of results of robustness testing experiments:

- Erroneous inputs adopted by robustness testing are not guaranteed to be representative of real faults [MBD+06] (i.e. we do not know if a real residual defect would produce such an erroneous input), and they can not be directly mapped to software defects (i.e. we can not make assumptions on the particular defect that produces that input). However, the injection of exceptional inputs can be viewed as the assignment of a wrong value to a variable, so we assume that they are representative of the Assignment and Interface ODC classes (these classes are defect types of a significant part of real defects, e.g. MVAE, WVAV, MVI, MVAV, WAEP, and WPFV in figure 3.3);

- Robustness testing does not enable to test the effects of those errors that propagate in a different way than function parameters or return value; because the Linux kernel is monolithic, device drivers share the same address space of other kernel components, and thus it is susceptible of "overlay" errors that corrupt the memory (see [SC91]). Nevertheless, we observed in our experiments that error injection at the interfaces can produce failure manifestations similar to the ones due to memory corruption (e.g. if a pointer is changed, the kernel reads the contents of a wrong memory region, uncorrelated with the correct one).

To define robustness testing experiments, we have to identify the interfaces of the BT to other components in the SUB; in the case of the Linux operating system, we have two distinct interfaces dedicated respectively to applications running in user-space (represented by the system calls) and to device drivers (represented by internal kernel functions). Since application-level stressing using Ballista did not discover robustness failures, we focus on the interface to device drivers. Both a device driver and the kernel provide several services in the form of explicitly exported functions, and each one make use of the services provided by the other: to correctly emulate software faults activated in the drivers' code, we consider for error injection the input parameters of those functions exported by the kernel and invoked by the drivers, and the return values of the functions exported by the drivers and invoked by the kernel.

Among the different ways to inject errors through values passed from a driver to the kernel (bit-flip, data type error, fuzzing), we choose bit-flip: as described in [JSM07], it is the most performant in terms of number and severity of identified robustness failures, at the cost of an higher execution time for experiments, and due to the determinism of injected values it does not add more uncertainty to the outcome of experiments.

3.3 Dependability benchmarking of logging facilities

3.3.1 Definition of the experiments

In order to formalize the experimental assessment of the operating system under study, we consider the more general framework of dependability benchmarks; although we do not aim to compare different operating systems by their detection facilities, the measures of our interest fit well into dependability attributes of a system, thus they may be the subject of a dependability benchmark; therefore we impose to experiments the standard requirements defined for dependability benchmarks, such as the repeatability of the results. After the definition of the dependability benchmark, we illustrate it by describing and executing an implementation for the Linux kernel. In the following the dependability benchmark is briefly described in respect to the most relevant aspects impacting on a benchmark (see figure 1.6):

Categorization

- This dependability benchmark is conceived to assess the failure detection capabilities of COTS operating systems, under workload and faultload conditions typical of a critical context with dependability requirements.
- The benchmark is aimed to COTS system designers interested to develop fault tolerance and recovery strategies (e.g. fault diagnosis).

Measures

- The measures evaluated by the benchmark are the coverage and the latency of the failure detection mechanisms of the operating system, as defined at the beginning of this chapter; these measures are related to dependability and specific of a particular facet of the system, and should be quantitatively assessed.
- The measures have to be evaluated by experimentation; error injection is employed to emulate the occurrence of the activation of a fault.

Experimentation

- The coverage in respect to drivers' failures (which are represented by errors injected) is estimated by the ratio between the number of experiments in which the presence of the error is notified, and the total number of experiments. Instead, the coverage in respect to system-level failures (see 3.1) is estimated by the ratio between the number of experiments in which a failure of the system is observed and a log is produced, and the total number of system-level failures.
- The latency in respect to drivers' failures is estimated by the interval that ranges from the time in which the error is injected in the BT, to the time of the first log notification of that error. Because we often are not able to quantify the time in which a failure of the system as-a-whole occurs, we estimate the latency in respect to system-level failures using the latency in respect to drivers' failures: it is an upper bound (because a system-failure is caused by a failure of a driver), and it is tight to the real latency value (we observed in our experiments that, when there was a system-level failure, e.g. a crash or an hang, it occurred shortly after the injection).
- The faultload assumed for experimentation is represented by erroneous values passed at the interfaces between the FIT and the BT.

This dependability benchmark aims to evaluate the suitability of detection of an operating system to be employed in a critical context, therefore a representative workload should be adopted to draw realistic conclusions from the experiments. In this chapter, we still employ a synthetic workload in order to simplify the illustration of the implementation of the benchmark; moreover, these experiments can provide an overview of the kernel's failure modes, increasing our understanding of the system, and an initial quantitative evaluation of the failure detection mechanisms.

As we previously stated, robustness testing on the Driver Programming Interface between the kernel and the drivers constitutes the faultload of the benchmark, where the FIT is represented by the device drivers, and the kernel is the BT. Thus, it is necessary to identify all the functions actually called by the FIT and the BT when the considered workload is executed; to shorten the total execution time of the benchmark, the target functions may be the ones most often invoked.

Once an error is injected, we expect that the SUB will behave in different ways, that we classify in the following categories:

Crash the operating system is no more able to provide its services to applications; typically an explicit warning is prompted, and a software reboot is still possible;

Hang the operating system is running, but does not provide useful work for the applications; an hardware reboot is needed;

Workload failure an application is not able to conclude its work, or it produces erroneous results;

No signal no effect on the workload and on the operating system is observed after the conclusion of the experiment.

It should be observed that, even if an experiment leads to a crash or an hang, log notifications can still be produced before and during the interruption of service of the SUB. If the operating system executes for the whole experiment duration without crashing, the desired behavior is the production of a log message about the presence of a problem (the error injected), whereas it affects or not the workload; if the error goes unnoticed, or the error is silently tolerated, the experiment highlighted a *failure of logging mechanisms* (the system did not produce a log message when it should have). We also observe that a crash or a hang may lead or not to a workload failure (e.g. the system may crash after that the workload correctly executed); thus we classify as a workload failure only those cases in which the operating system is still working, although its internal error propagated to the workload through its services' interfaces.

In addition to the experimental outcomes previously listed, we are also interested in measuring the time spent to execute the workload, in order to observe whether the injection of a fault has an effect on the performance of the workload, as well as on its correctness.

To emulate software faults in a driver's source code, errors are injected by bit-flipping values in procedural invocations in the kernel. The injection is triggered when the target function is called after a predefined amount of time since the experiment start time: this is both a spatial and temporal criterion (the error is injected *at the first function invocation*, and *after a predefined instant*). Because of the complexity of the SUB, we can not guarantee that an exact sequence of instructions will be executed by the target machine, and so fault injection experiments are not deterministic; nevertheless, we monitored the execution of the kernel by tracing relevant internal events timestamps, and we evaluated the covariance between each temporal sequence of the same event type: we observed a strong similarity, and thus we argue that our experiments are statistically reproducible. In order to validate this claim, experiments should be executed more than one time and have to produce non contrasting results between different runs; the repeatability of the results of our experiments is discussed in-depth in section 3.3.4. It should be noted that neither a spatial-only injection criterion (a fault is triggered when a particular instruction is executed) guarantees determinism because of the un-controllability of the execution flow.

3.3.2 Experiments setup

We consider a workload that makes use of both non-hardware and hardware related services (e.g. I/O requests to disks, network cards) , in order to reproduce stressing work conditions and to stimulate different kernel entry points. The experimental scenario is composed by a server and a client machine connected by a local area network: the server constitutes the SUB, in which the Linux kernel (the BT) is executed; its configuration is described in table 3.1. The Ballista testing tool (included in the Linux Test Project suite, [12]) is used as application-level workload: it is composed of some C++ programs that invoke system calls with predefined input parameters, the GNU C compiler (the test suite is partially re-compiled for each system call tested) and other UNIX utilities. Because of the brevity of each experiment, only a narrow subset of system calls is tested by Ballista: remember that we do not execute a system call robustness test, but we conduct fault injection inside the operating system, so any application that frequently invokes kernel facilities (in particular, in order to increase the fault activation rate, kernel services related to fault-injected parts have to be invoked, i.e. device drivers) is sufficient for our purposes; if it is required an evaluation of how the system will behave under faults in a real context, a more realistic workload is needed (e.g. the system will not be affected by a fault occurred in a kernel component never activated; in other words, the fault is not triggered).

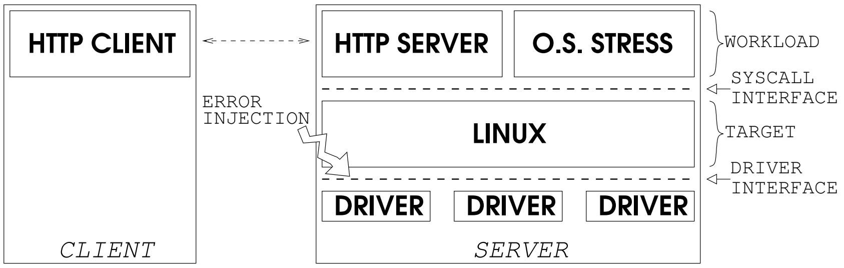


Figure 3.5: *Setup of error injection experiments.*

Table 3.1: *Hardware and software configuration of the System Under Benchmark, employed for error injection experiments (HP xw4400 Workstation).*

CPU	Intel Core^{TM} 2 Duo E6600 2.40GHz
RAM	4 GB
Disk controller	Integrated Intel Corporation 82801GR/GH SATA 3 Gb/s interface
Hard disk	250 GB SATA 3 Gb/s NCQ (7200 RPM)
Network card	Integrated Broadcom 5755 NetXtreme Gigabit Ethernet PCI Express
Linux kernel version	2.6.21
Linux distribution	Red Hat Enterprise Linux AS release 4 (Nahant Update 2)
Apache webserver	2.0.52
Ballista (LTP)	snapshot 20070630
SystemTap	snapshot 20070721

Moreover, the application workload is composed by the Apache webserver; a client machine downloads several files from the server (about half a gigabyte), thus stressing the I/O and network subsystems; again, a more realistic network workload should be employed to assume the results as representative of the system in the operational phase. To verify the correctness of the workload execution, we inspect the Ballista test logs to identify robustness failures, and we make a checksum verification on data downloaded by the HTTP client; if an error is found on application logs or by a checksum, or if the forced or wrong exit of a process occurs, the experiment's outcome is classified *workload failure*.

We choose as FIT for the dependability benchmark the device drivers of the available serial ATA disk and Ethernet network card: they are called *ahci* and *tg3* respectively; moreover, the *libata* component (a group of common routines for serial ATA drivers) is considered as part of the disk driver. In the following, we are going to refer to them as the *sata driver* and the *network driver*, respectively.

Device drivers interact with other kernel components through procedure invocations; they may (but do not have to) be compiled as kernel modules, and loaded at runtime. The interface between kernel and drivers is made by the set of functions defined by the drivers and called by the kernel, and vice versa. In the Linux 2.6 kernel tree, the functions of the kernel that can be called by a module (and vice versa) should be explicitly exported to be correctly resolved and linked; in some cases, there are functions not explicitly exported that are invoked in a component by first retrieving a pointer to the function's code (e.g. the pointer of a routine for I/O requests handling, defined in a device driver, is passed to the kernel when the driver is initialized; when the kernel has a request to dispatch to the driver, it invokes the function which that pointer refers to).

To identify locations for error injections, the functions constituting the interface between the kernel and the target device drivers should be identified (figure 3.6). To find invocations of kernel's functions by the driver, we employed static source code analysis on exported function symbols; to find invocations of driver's functions, we considered both static analysis and dynamic probing of all the functions defined in the driver, because of the pointer-based invocation mechanism employed by the kernel.

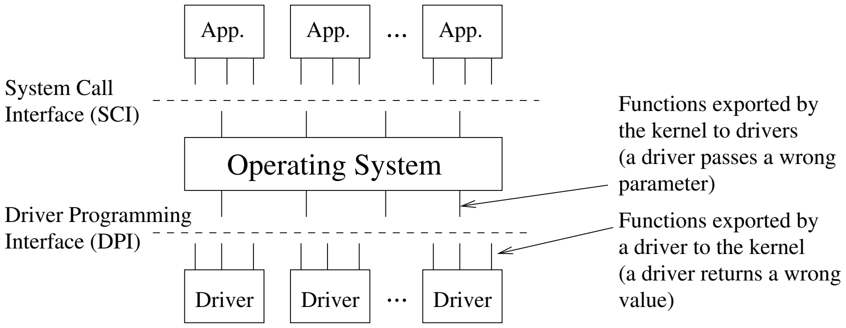

App.
App.
...
App.
System Call Interface (SCI)
Operating System
Driver Programming Interface (DPI)
Driver
Driver
...
Driver
Functions exported by the kernel to drivers (a driver passes a wrong parameter)
Functions exported by a driver to the kernel (a driver returns a wrong value)

Figure 3.6: *Interfaces between the kernel and device drivers, at which errors are injected.*

After the injection locations were found, they should be monitored during the execution of the selected workload; the functions that will be the target of the injections are chosen such that they account for the most part of the invocations, and that the total number of injection experiments is significant (e.g. roughly a thousand). In our implementation, 544 bit targets in 17 parameters of 10 functions were identified for the network driver, and 160 bit targets in 5 parameters of 5 functions for the sata driver. Finally, error injection experiments can be accomplished by targeting selected functions.

In order to implement function monitoring and error injection, we employed dynamic probes: they are already provided by the mainline kernel (see KProbes and SystemTap, § 2.5), they are a flexible, generic and simple-to-use tool, and they do not require modifications to the source code of the target modules, thus speeding up the execution of experiments and reducing the intrusiveness of the experiment setup in the SUB.

In both injection (see listing A.1) and monitoring (see listings A.2 and A.3), dynamic probes were placed at the entry points of kernel's functions called by a module, and at the exit points of module's functions called by the kernel. When a probe is hit by the execution flow, a check is made to ensure that the kernel function was actually called by the target module in the first case, or that the module function was not called by the module itself in the second case: to do this, the kernel stack is inspected and the caller of the probed function is located using the return address (see Appendix A for a detailed description).

To measure the latency of logging mechanisms, dynamic probing was employed again (see listing A.5); a KProbes module (implemented with SystemTap), probes for invocations of the `sys_write` system call by the user-space process `syslogd`, which collects and stores log messages from the kernel and other parts of the operating system. It should be noted that the KProbes modules for injection and for measurements can execute concurrently in kernel space. The timestamp of the write is then saved on the disk. Although we can obtain timestamps in nanoseconds, we consider only digits up to microseconds, because of the latency between the issue of a software interrupt and its handling. The log latency is given by the difference between the injection timestamp (saved by the injector module) and the logging timestamp. By using the same dynamic probing mechanism for error injection (and timestamping) and log timestamping, we reduce the alteration to the estimated log latency introduced by the software interrupt latency (the difference between these two measurement should mitigate such alteration).

Because `syslogd` and the KProbes modules are part of the SUB, they are both exposed to error propagation, therefore it is not guaranteed that an error notification by the kernel will be logged, or that the logging latency will be available. In the latter case, we estimate the latency using the timestamp entries in `syslogd`'s messages, with a lower precision (log messages are timestamped in seconds); in general, an approximation should be conservative and should not underestimate the latency and the coverage (e.g. it should round up to the nearest integer rather than to round down). Moreover, it is important to not consider those log messages not related to failures when computing log coverage and latency (for example, messages inherent to ordinary system maintenance such these produced by the `crond` daemon).

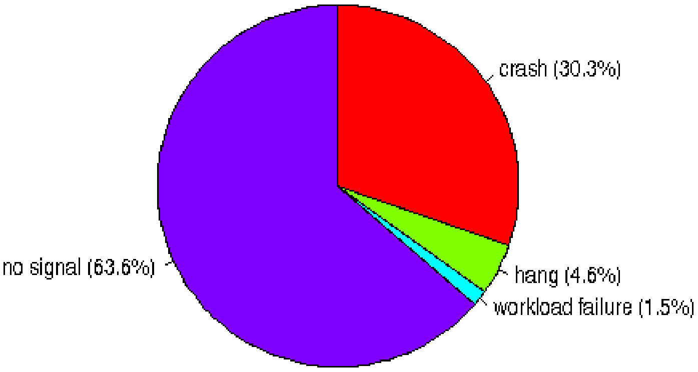
In order to prove the repeatability of the benchmark, it should be executed two or more times; the experiment is considered repeatable if its results (execution outcomes, log coverage and latency, and experiment duration) are very similar between repetitions. To ensure such property, the experiments should be completely automated (e.g. no human intervention should be scheduled); in our benchmark implementation, all the experiments are automatically executed by a script running after the boot sequence (the only human intervention required is an hard reboot in the case of a hang outcome).

3.3.3 Results

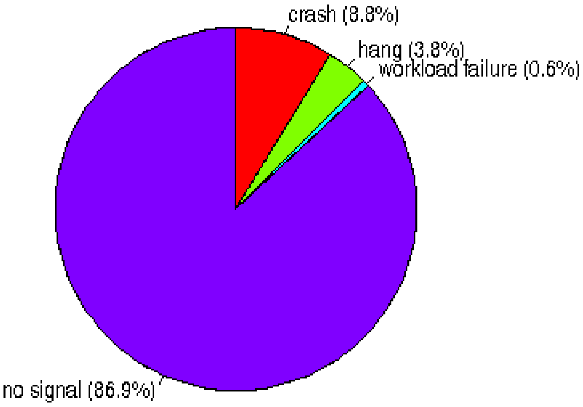
We here summarize the results obtained by the dependability benchmark execution, and defined in the previous subsection. In figures 3.7a and 3.7b, the percentages of the outcomes observed for the workload and the operating system are represented. The first consideration is that in the most of cases the injection of an error did not lead to an observable failure: three explanations of this outcome are:

- The error was tolerated by the operating system, e.g. it was identified by an internal check or it was overwritten (transitions in the "good" state in figure 3.4);
- The error is dormant and was not activated during the experiment, and thus it can lead to failures in the future (the "latent error" state in figure 3.4);
- The operating system failed in providing a correct service to applications, but the workload was able to tolerate the failure.

In a small number of cases, we obtained workload failures when injecting errors in both drivers. In these cases, the operating system was still working and the workload was completed, but an erroneous result was produced (e.g. process killing or exit, errors in application logs or found by a checksum); in particular, we observe that in some cases the robustness test at the system calls (i.e. Ballista), which in normal conditions does not discover robustness failures, fails when an error is injected at the drivers' interface. Although the occurrence of incorrect workload result is unlikely, it is an evidence that an unsafe behavior of the operating system is possible in the presence of an error state; nevertheless, further experiments with a workload and a faultload representative of a critical context should be executed to draw conclusions about the safety attributes of the operating system.



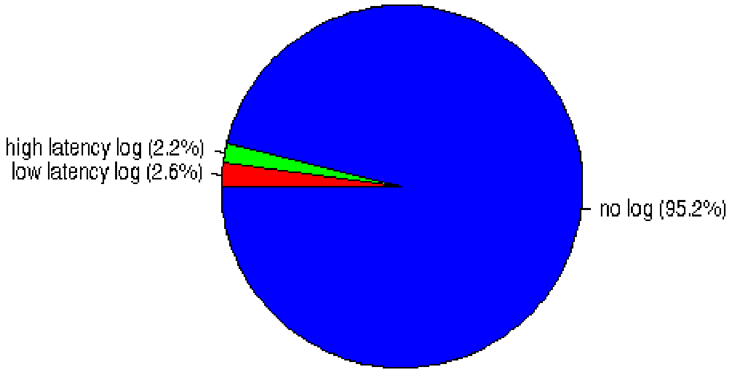
(a) network driver.



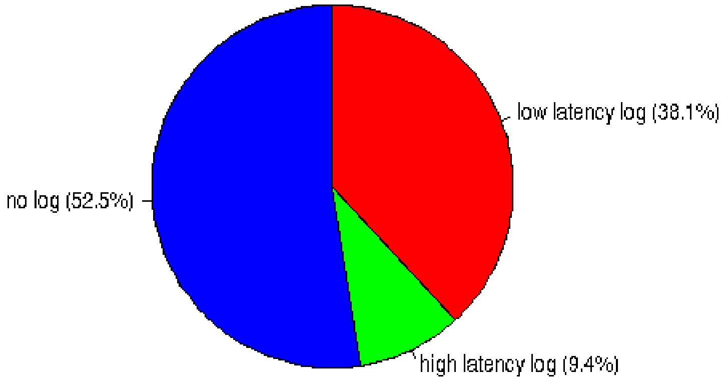
(b) sata driver.

Figure 3.7: *Outcomes observed after error injections at the device driver interface.*

In figures 3.8a and 3.8b, the estimated coverage (in respect to drivers' failures) of the operating system logs is displayed. We observe that for both the drivers there were a majority of cases in which no log at all is produced. Moveover, error injections in the sata driver lead to a much higher detection percentage than the network driver; it can be explained by the fact that more numerous and structured kernel subsystems depend on the disk driver (e.g. the SCSI layer – the SATA port is seen as a SCSI bus –, the Block I/O layer and the filesystem), and thus a more strict checking is made on the operations based on that driver.



(a) network driver.



(b) sata driver.

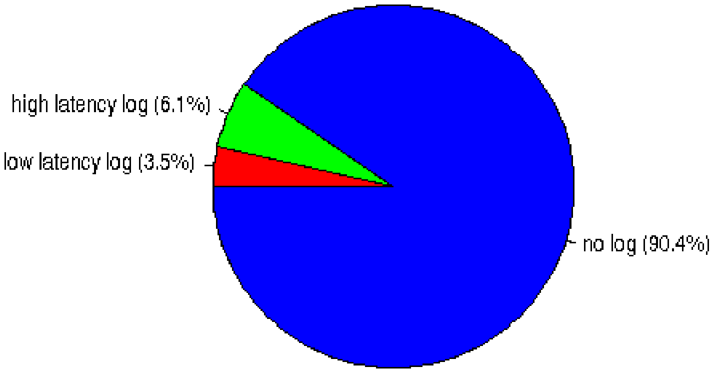
Figure 3.8: *Log coverage measured after error injections at the device driver interface.*

We further differentiate between cases in which the log latency was low (i.e. less than 1 second), and cases that lead to an high log latency. In fact, for both drivers there is a remarkable subset of cases in which the latency is very low (ranging from a few of milliseconds to hundreds of milliseconds); it is due to the fact that some data exchanged with device drivers is immediately checked by the kernel, and therefore the log latency is composed for the most part by the I/O delay needed to write the log to the disk. In the rest of cases, the error was notified much later (the order of magnitude of this delay is tenths of seconds), and log entries were referred to several parts of the kernel code and to the workload (e.g. a process was killed), thus we conclude that the errors propagated in different components of the SUB. In table 3.2 the measured mean latencies for both drivers are quantified.

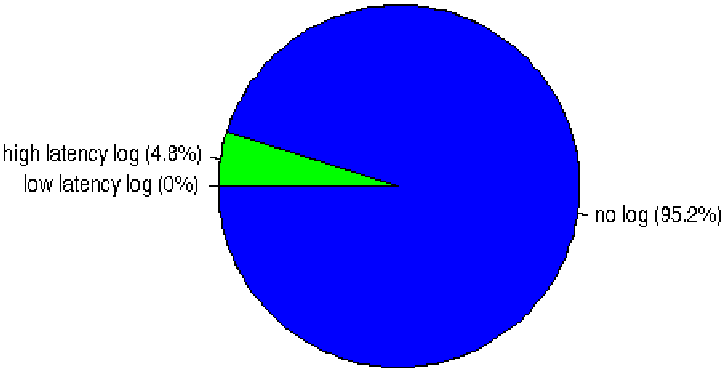
Table 3.2: *Measured log latencies for the Linux kernel after error injections representative of faults in two device drivers. We differentiate between low (i.e. less than 1 second) and high latencies.*

	network	sata
Mean low latency	0.104590 s	0.004791 s
Mean high latency	36.833334 s	29.627197 s

In figures 3.9a and 3.9b, it is shown the log detection coverage restricted to system-level failures, which is obtained by only considering experiments in which there was a failure of the system as a whole (i.e. the *crash*, *hang* and *workload failure* outcomes, accounting for 29.5% and 13.1% of network and sata drivers' failures, respectively). In the case of the injections at the network driver interface, the system-level coverage is slightly greater than drivers' failures coverage, because those experiments which were logged, did often lead to system-level failures; nevertheless, the coverage still results much less than the percentage of non-logged experiments. In the case of the sata driver, the system-level coverage is significantly smaller than drivers' failure coverage: a remarkable amount of system-level failures was not logged. This observation on the system-level coverage implies that the operating system has an insufficient ability to log those failures which were not tolerated, rather than to log tolerated ones.



(a) network driver.



(b) sata driver.

Figure 3.9: *Log coverage for system-level failures measured after error injections in the device driver interface.*

Finally, we report on measured execution times for the network workload (i.e. the time spent by HTTP clients downloading); these values are listed in table 3.3. We observe that in the case of the network driver there was not a significant deviation from the non-faulty behavior, but in the case of the sata driver there were several cases in which the download time considerably increased (about +50% compared with non-faulty behavior), and thus a performance decrease can be considered as a symptom of the presence of an error.

Table 3.3: *Total download times measured by the client in the case of no injection and injection in the drivers respectively (cases in which downloads were interrupted by a system crash are not considered).*

	no fault	network	sata
Mean	62.5625 s	61.765073 s	66.871486 s
Std. dev.	0.957427 s	1.086329 s	15.854937 s

3.3.4 Statistical validation

As discussed in previous sections, the experiments defined in the dependability benchmark are affected by uncertainty, i.e. the outcome of an experiment may vary between subsequent repetitions, due to the high complexity of the considered system. Therefore, the repeatability property of the dependability benchmark should be verified, so that obtained results can be trusted by a third party, and that the methodology can be used as a common way for users to evaluate logging facilities. A dependability benchmark is considered repeatable if *it guarantees statistically equivalent results when it is run more than once in the same environment* [MKA+01].

In order to evaluate the repeatability property, the experiments previously described were repeated for a second time. The observed distributions for outcomes (figure 3.7) and logging of tests (figure 3.8) are shown in tables 3.4 and 3.5. The χ^2 hypothesis test on the distributions of the two sets of experiments (*Set 1* and *Set 2*, respectively) was made, in order to confirm the following *null hyphotesis*:

The statistical distributions of the results are independent from the considered set of experiments.

An hypothesis test requires that a *test statistic* (i.e. a function $f(\overline{x})$ of the experimental results $\overline{x}$) is defined; then, experiments are executed, and the value of the test statistic is computed from the obtained statistical sample. If the value has *a very low probability of occurrence if the null hypothesis* *is true*, then we can reject the null hypothesis because it is contradicted by an experimental evidence. The *significance level*, indicated with $1 - \alpha$, is the probability to correctly reject the null hypothesis if the value of the test statistic occurs with probability α (e.g. $\alpha = 0.05$). Instead, we are reasonably sure that the null hypothesis is true if it can not be rejected, with an α low enough.

In our case, we have to verify that the probability that an experiment falls in one of a set of categories (*crash*, *hang*, *workload failure*, and *no signal* for outcomes; *low latency*, *high latency*, and *no log* for logging) is independent from the set of experiments (*Set 1* or *Set 2*). As prescribed by the χ^2 hypothesis test, we use as test statistic the following:

$$K = \sum_{i=1}^{M} \sum_{j=1}^{N} \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \tag{3.1}$$

in which we assumed:

- M is the number of categories (4 for outcomes, 3 for logging);
- N is the number of statistical samples (the 2 sets of experiments);
- O_{ij} is the number of experiments of the set j which fall in the ith category;
- E_{ij} is the expected number of the experiments for the ith category and the jth set, *given that the null hypothesis is true.*

The values E_{ij} are the joint probability[1] of occurrence of the ith category and of the jth set; because we made the hypothesis that sets and categories are *mutually independent*, then E_{ij} is the product of the *marginal probabilities* of i and j. For example, in table 3.4a, the probabilities of the *crash* category and the *Set 1* are $\frac{337}{1088}$ and $\frac{544}{1088}$, respectively; then, the expected number of experiments in *Set 1* which lead to a crash is $1088 \cdot \frac{337}{1088} \cdot \frac{544}{1088} = 168.5$ (table 3.4b), which is compared to the observed number 165 (table 3.4a).

[1] To be more precise, the joint probability is the ratio E_{ij}/n, where n is the total number of experiments; instead, E_{ij} is the *absolute* number of experiments of the pair set/category.

Table 3.4: *χ^2 test for statistical reproducibility of outcomes distributions.*

(a) *Observed distribution for tests on the network driver ($\nu = 3$, $K = 2.50$, $\chi^2_{0.05} = 7.82$).*

	crash	hang	w. failure	no signal	
Set 1	165	25	8	346	544
Set 2	172	23	3	346	544
	337	48	11	692	1088

(b) *Expected distribution for tests on the network driver.*

	crash	hang	w. failure	no signal	
Set 1	168.5	24	5.5	346	
Set 2	168.5	24	5.5	346	

(c) *Observed distribution for tests on the sata driver ($\nu = 3$, $K = 2.37$, $\chi^2_{0.05} = 7.82$).*

	crash	hang	w. failure	no signal	
Set 1	23	14	1	122	160
Set 2	15	17	2	126	160
	38	31	3	248	320

(d) *Expected distribution for tests on the sata driver.*

	crash	hang	w. failure	no signal	
Set 1	19	15.5	1.5	124	
Set 2	19	15.5	1.5	124	

Table 3.5: *χ^2 test for statistical reproducibility of logging distributions.*

(a) *Observed distribution for tests on the network driver ($\nu = 2$, $K = 6.44$, $\chi^2_{0.05} = 5.99$).*

	low latency	high latency	no log	
Set 1	14	12	518	544
Set 2	4	17	523	544
	18	29	1041	1088

(b) *Expected distribution for tests on the network driver.*

	low latency	high latency	no log	
Set 1	9	14.5	520.5	
Set 2	9	14.5	520.5	

(c) *Observed distribution for tests on the sata driver ($\nu = 2$, $K = 0.48$, $\chi^2_{0.05} = 5.99$).*

	low latency	high latency	no log	
Set 1	60	30	70	160
Set 2	58	35	67	160
	118	65	137	320

(d) *Expected distribution for tests on the sata driver.*

	low latency	high latency	no log	
Set 1	59	32.5	68.5	
Set 2	59	32.5	68.5	

Because the results of an experiment are random variables, then the test statistic K is also a random variable. It can be shown that, for an high enough number of experiments, K represents a χ^2 *random variable with* $(N-1)\cdot(M-1)$ *degrees of liberty*, of which we known the probability distribution. Then, if a given experiment produces $K=k$ such that $\mathcal{P}(K \geqslant k) \leqslant \alpha$, then we reject the null hypothesis with a significance level of $1-\alpha$.

In all cases of tables 3.4 and 3.5 (except for logging for the network driver), the null hypothesis can not be rejected, with a significance level of 0.95, because we have that $K < \chi^2_{0.05}$ in each case; this means that the hypothesis of independence (and therefore the repeatability of the results) is reasonably verified. Instead, in the case of logging for the network driver, the hypothesis of independence can be rejected with a significance level of 0.95, because $K = 6.44 > \chi^2_{0.05} = 5.99$ (table 3.5a). Nevertheless, because $K = 6.44 \simeq \chi^2_{0.04}$, the hypothesis can not be rejected with a significance level of 0.96. This non negligible difference between the observed and the expected distributions is due to the portion of experiments which were classified as *low latency* in *Set 1*, and as *no log* in *Set 2* (instead, there were observed crashes); because of the small number of cases of *low latency* for both sets, the χ^2 test was noticeably influenced by this difference. Moreover, the small number of experiments for the *low latency* category weakens the initial assumption about the probability distribution of K.

However, if the categories *low latency* and *high latency* are merged in one (table 3.6), the coverage distribution results independent from the set in the case of logging for the network driver, because the null hypothesis can not be rejected. Therefore, we conclude that the coverage is reproducible, although it was not possible to reproduce the sub-categories *low latency* and *high latency.*

We can not make assumptions on the probability distributions of the statistical means used for evaluation of the mean low and high latencies, because we do not know the distribution of the single experiment, and the independence[2] between experiments does not seem to hold (because, as we will see

[2] The hypothesis of independent and identically distributed random variables is required for the Central Limit Theorem to assume that the mean value of several experiments is normally distributed.

Table 3.6: *χ^2 test for statistical reproducibility of coverage distributions.*

(a) *Observed distribution for tests on the network driver ($\nu = 1$, $K = 0.56$, $\chi^2_{0.05} = 3.84$).*

	log	no log	
Set 1	26	518	544
Set 2	21	523	544
	47	1041	1088

(b) *Expected distribution for tests on the network driver.*

	log	no log	
Set 1	23.5	520.5	
Set 2	23.5	520.5	

(c) *Observed distribution for tests on the sata driver ($\nu = 1$, $K = 0.11$, $\chi^2_{0.05} = 3.84$).*

	log	no log	
Set 1	90	70	160
Set 2	93	67	160
	183	137	320

(d) *Expected distribution for tests on the sata driver.*

	log	no log	
Set 1	91.5	68.5	
Set 2	91.5	68.5	

in section 5.4, experiments on the same parameter have an high probability to produce the same outcome). Therefore, we do not apply an hypothesis test to evaluate the reproducibility of measured latencies; nevertheless, the latencies are not very different in terms of relative differences (figures 3.7 and 3.8), except for network driver's mean low latency, of which the cases were too few to draw a reliable estimation (see table 3.5a). Therefore, we conclude that also latency was reproducible across different sets of experiments.

Table 3.7: *Comparison between mean low latencies for different experiments sets.*

	Set 1	Set 2	Difference
network driver	0.104590 s	0.067330 s	-35.6 %
sata driver	0.004791 s	0.004344 s	-9.3 %

Table 3.8: *Comparison between mean high latencies for different experiments sets.*

	Set 1	Set 2	Difference
network driver	36.833334 s	38.958075 s	+5.8 %
sata driver	29.627197 s	29.317801 s	-1.0 %

3.4 Conclusions

In this chapter, we cope with the problem of the assessment of the failure detection ability of commodity operating systems, in terms of log coverage and latency. We first discuss different ways to artificially accelerate the activation of software faults in operating systems by emulation. In particular, we take in account the G-SWFIT technique and the robustness testing approach, and we argue that the latter is more suited for our purposes.

Thereafter, we propose a dependability benchmark specification based on robustness testing at the interface between device drivers and the operating system kernel, in which an error is injected by bit-flip of values exchanged between a device driver and the kernel. This benchmark is then implemented for the Linux operating system, and obtained results are reported.

The experiments highlighted that for the majority of injected errors there were not logs that notified the presence of the error in the operating system (the estimated coverage is less than 50%). Moreover, whenever logs were emitted there was a non negligible part of cases in which the measured latency was very high (more than 1 second), leading to error propagation and incorrect or missing information about the faulty component, and several non-logged injections lead to severe system-level failures. Thus, more effective mechanisms for error logging are needed in order to increase the detection ability of the operating system, that can be the basement for fault tolerance and recovery mechanisms.

CHAPTER 4

A technique for failure detection

> Much of the excitement we get out of our work is that we don't really know what we are doing.
>
> Edsger W. Dijkstra

4.1 Introduction

In the previous chapter, the Linux operating system's robustness, in respect of faults in its device drivers, was evaluated. Moreover, it is pointed out that in the majority of cases there were not log entries with a description of the problem, or at least a hint about its presence in the system; when log entries were produced, a significant amount of time often elapsed from the activation of the fault. In the following, we introduce a new approach for the detection of failures, based on the monitoring of the activities of device drivers, to produce useful hints for off-line and on-line analysis of outages' causes and to offer support for fault treatment techniques. The proposed approach is based on empirical observations of the behavior of the system under faults, using the data collected during error injection experiments.

4.2 Considerations on experimental data

In the implementation of the dependability benchmark described in the previous chapter, we also included a kernel monitoring module for tracing I/O activities (*events*) of the network and disk devices drivers (see listing A.6), in terms of type and time of transferred packets and data blocks read/written from/to disks. As highlighted in table 3.3, a performance degradation was actually observed in several cases; there were cases in which the system apparently behaved well (e.g. the data transmitted by the web server was not corrupted, and workload executed without errors), although the test duration was higher than normal (e.g. the workload terminated after 90 seconds, compared to a duration of about 60 seconds when no error was injected). When workload execution time was significantly greater, the traces of drivers' events also differed from the ones in non-faulty conditions; in particular, the throughput of events' occurrences (defined as the ratio between the number of events occurred in a time unit, and the time unit itself) experienced sudden changes during the execution, after that an error was injected.

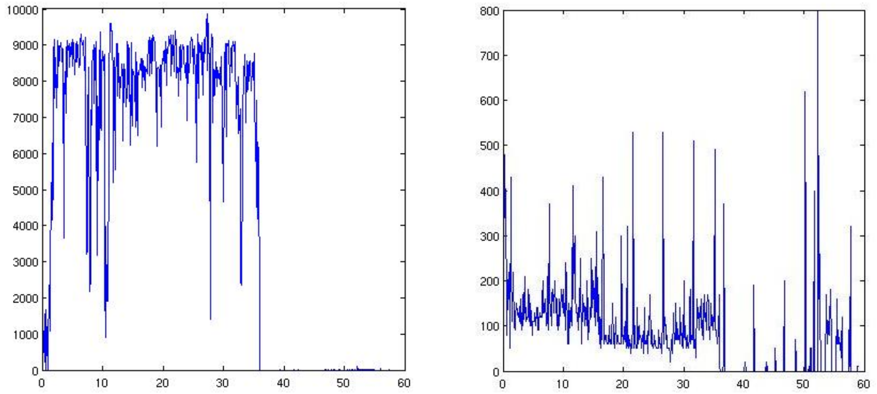
In figures 4.2 and 4.3, we observe non-negligible time periods in which drivers activities were stall (several consecutive throughput samples are equal to zero): there are abrupt decreases of the throughput after the injection of an error, followed by other changes (we refer to them as *throughput losses* and *peaks*); such behavior was not observed when no errors were injected (figure 4.1), in which drivers activity goes uninterrupted until the workload completion.

In order to define an approach for the detection of system's failures, we investigated the relationship between the correctness of system's behavior and its performances. In the following, we are going to:

- Make further fault injection experiments, to find the differences in the system's behavior when a failure exists, compared to the behavior in absence of failures;

- Formally express those differences;

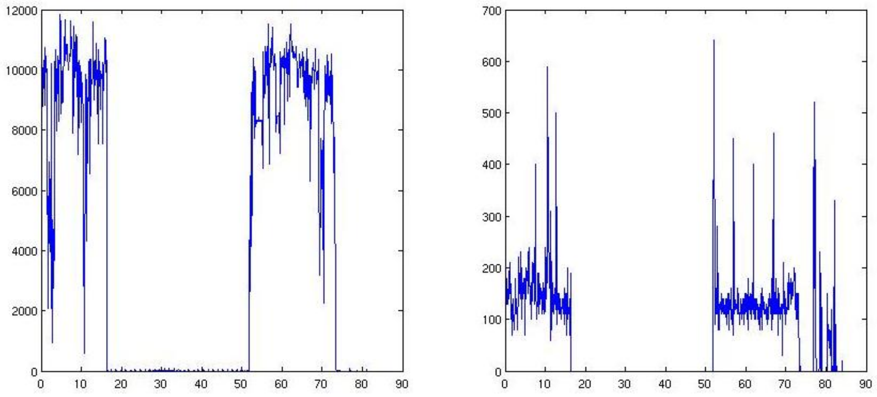
- Explain how differences can be highlighted at run time, although the



(a) *Packet transmission and reception.*

(b) *Block reads and writes.*

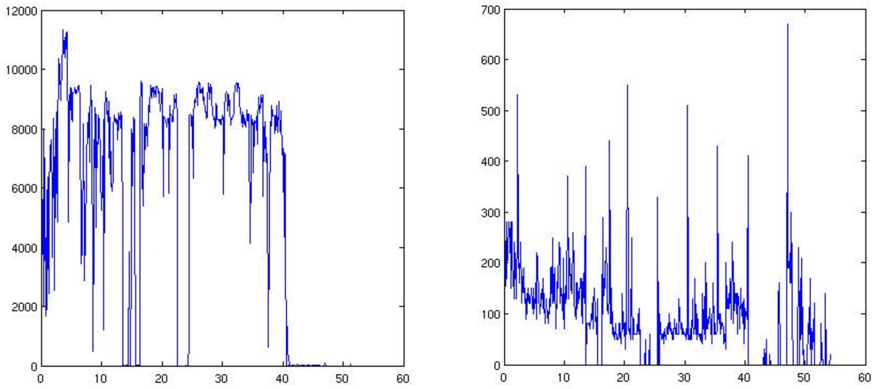
Figure 4.1: *Plots of throughput during the correct workload execution (sample period of T = 0.1 s).*



(a) *Packet transmission and reception.*

(b) *Block reads and writes.*

Figure 4.2: *Plots of throughput during an error injection experiment in the disk device driver (sample period of T = 0.1 s; the error was injected after about 10 s).*



(a) *Packet transmission and reception.*

(b) *Block reads and writes.*

Figure 4.3: *Plots of throughput during an error injection experiment in the network device driver (sample period of T = 0.1 s; the error was injected after about 10 s).*

correct behavior is not known *a priori* (the actual behavior is compared to the estimated *expected* behavior);

- Design an algorithm that detects such differences by on-line monitoring of system's execution.

To perform fault injection experiments, we used the fault injection framework included in the Linux kernel [29] [7], for injection of software implemented faults (SWIFI), that could be representative of hardware faults; an error was injected in the I/O block requests to the disk, by forcing the return of an error code (`-EIO`) from the `generic_make_request()` routine in the Block I/O layer (the kernel subsystem interposed between filesystems and device drivers), to emulate the occurrence of a transient fault in the disk (after the failure of a request, the following ones are correctly managed). Compared with software faults injected in the previous dependability benchmark, those faults are simpler and produce more predictable effects: they cause the immediate failure of a single I/O operation, thus the system is only affected at the time of injection, and the effects of error injection are immediately observable.

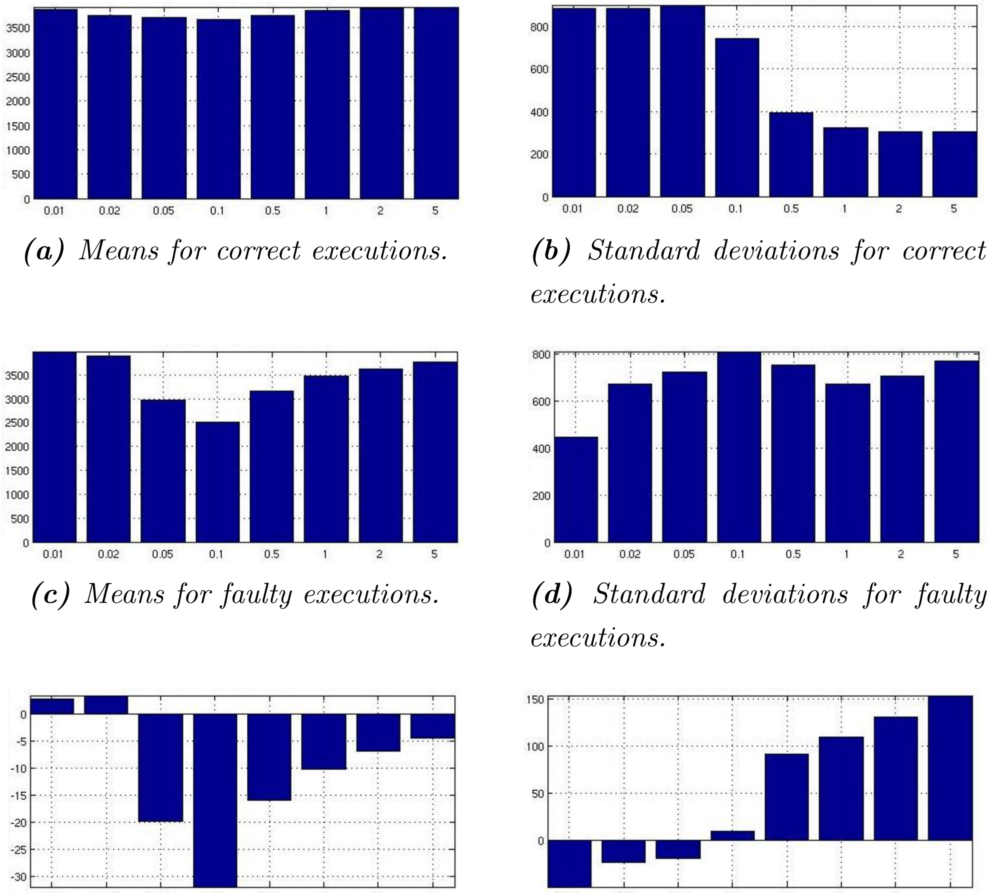
We injected ten consecutive request failures during the execution of the same workload of section § 3.3.2, at a predefined time, although the fault is activated only after the execution of the instrumented routine; the targeted requests occurred within a time interval of 0.1 seconds. We repeated the same experiment 25 times with fault injection, and 25 times without injections; we paid attention to not introduce significant differences between the faulty and non-faulty scenarios: the fault injection framework was activated in both cases, and the same user-space scripts were executed (they did not produce any effect in the latter case), in order to avoid systematic measurement errors. Several significant events of the operating system were monitored (using the LKET tapset library, § 2.5), together with the exact activation time of the instrumented routine.

Finally, we evaluated the mean and the standard deviation of the number of monitored events in time periods immediately following the activation of the faults. Among the events defined in the tapset library, we noticed an alteration in events related to I/O and device drivers. In figures 4.4, 4.5, and 4.6, it is showed the mean value and the standard deviation of those events for faulty and non-faulty experiments, evaluated in different time periods.

As expected, we see that the mean values differ within short periods (expressed as difference in percentage), and that faulty experiments show a greater variability (represented by a greater standard deviation), in particular for period lengths of 0.1 and 0.5 s; for example, in figure 4.4, there is a relative difference of 30% (figure 4.4e) between the mean number of transmitted packets in correct (figure 4.4a) and faulty executions (figure 4.4c), for a time window of $T = 0.1$ s. Thus:

Observation 1 : *there exists a difference in terms of sampled throughput from the expected behavior when a failure occurs.*

In this chapter, we refer to the *expected behavior* as the throughput most likely to be observed in a particular time period, if the system behaves correctly; the expected behavior is estimated by the mean value from several correct executions. In our experiments, the observed throughput differed from the expected one when a failure was induced. A failure may cause such differences because of operation repetitions (e.g. an user application repeats



(a) *Means for correct executions.*

(b) *Standard deviations for correct executions.*

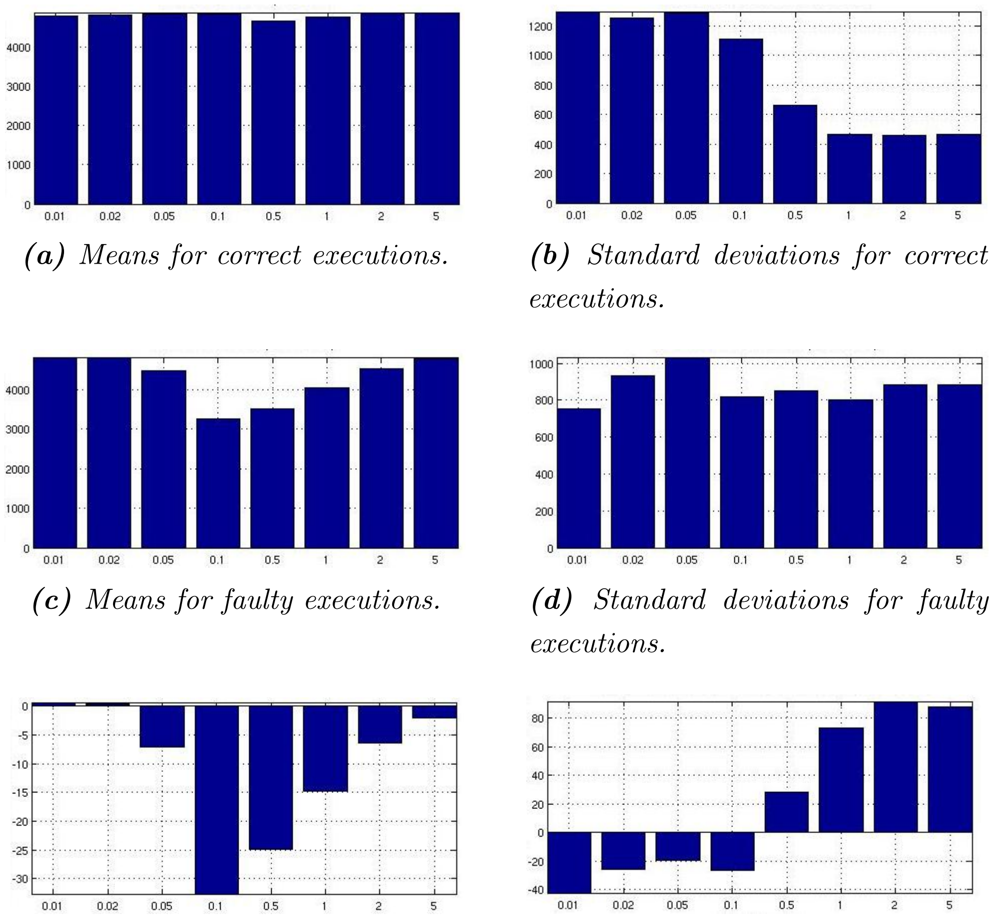
(c) *Means for faulty executions.*

(d) *Standard deviations for faulty executions.*

(e) *Difference in percentage between correct and faulty means.*

(f) *Difference in percentage between correct and faulty standard deviations.*

Figure 4.4: *Means and standard deviations of packet transmission for different period lengths (in seconds) after fault injection.*



(a) Means for correct executions.

(b) Standard deviations for correct executions.

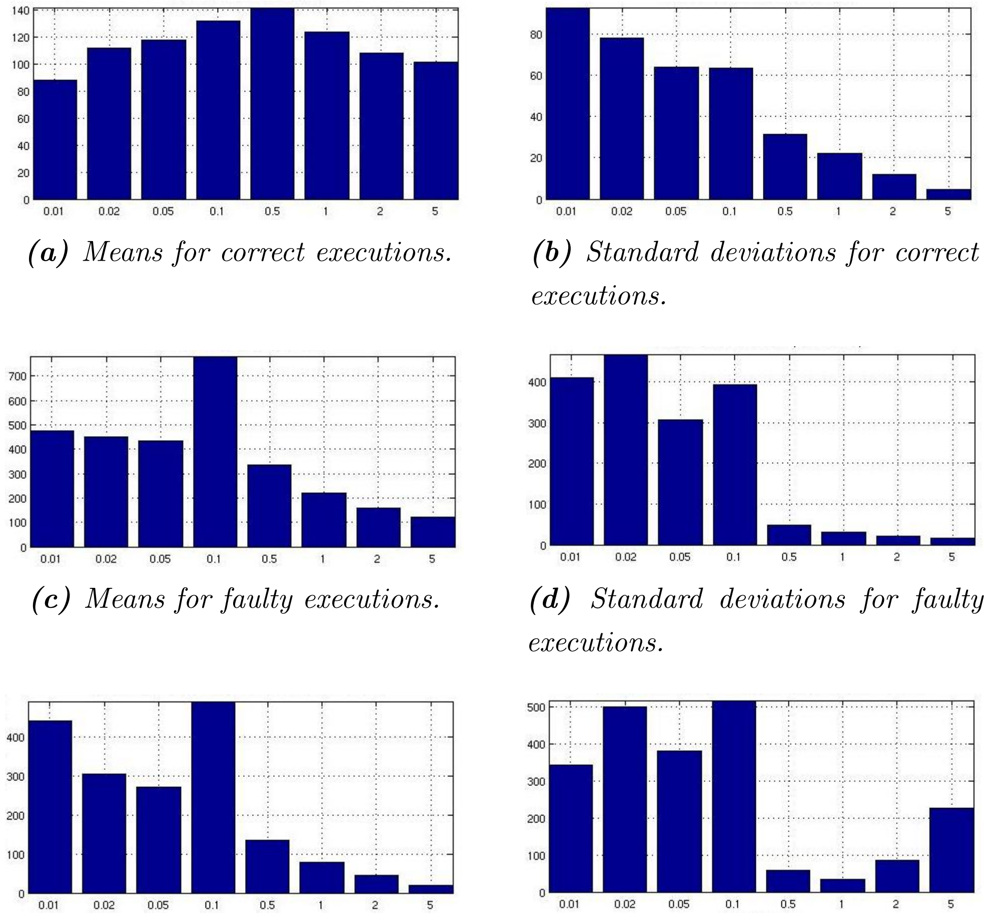
(c) Means for faulty executions.

(d) Standard deviations for faulty executions.

(e) Difference in percentage between correct and faulty means.

(f) Difference in percentage between correct and faulty standard deviations.

Figure 4.5: *Means and standard deviations of packet reception for different period lengths (in seconds) after fault injection.*



(a) *Means for correct executions.*

(b) *Standard deviations for correct executions.*

(c) *Means for faulty executions.*

(d) *Standard deviations for faulty executions.*

(e) *Difference in percentage between correct and faulty means.*

(f) *Difference in percentage between correct and faulty standard deviations.*

Figure 4.6: *Means and standard deviations of block reads and writes for different period lengths (in seconds) after fault injection.*

the access to a resource, or a reliable protocol such as TCP retries the transmission of a packet), waits for timeouts, exception handling, aborts, and so on. Moreover, the mean value for correct behavior within $T = 0.1$ s is well approximated by the mean value evaluated within $T = 5$ s (figure 4.4a); this property leads to the following statement:

Observation 2 : *the expected behavior can be approximated by averaging the observed behavior in a longer period, in which the workload does not vary noticeably.*

Because we make the failure disappear after the injection (subsequent I/O requests are correctly executed), the mean values observed in faulty and non-faulty executions for a time window of $T = 5$ s (figures 4.4a and 4.4c) are very close, therefore:

Observation 3 : *Even when there is a failure, the expected behavior can be approximated by the average throughput in a time period including non-faulty behavior.*

If we can compare the observed throughput with the expected behavior for that time period, we will point out a difference between them; because we know that a significant difference can be probably caused by a failure, this is a way by which we can detect the presence of a failure. Nevertheless, we do not know *a priori* the expected behavior; if the previous statements hold, we can approximate it by the throughput measurements collected in time periods next to the period in which a failure occurs:

Observation 4 : *the expected behavior can be approximated by the observed behavior under failure, if it is evaluated in a time period long enough.*

Using the considerations made in this section, we are going to define a technique for failure detection, based on hypotheses that hold in a large amount of cases, that can be used in a feasible way for on-line monitoring of critical systems, in which the timely detection of the presence of a problem is a key prerequisite to ensure an high degree of reliability and safety.

4.3 Proposed algorithm

In order to exploit the knowledge about the behavior of the system under failure, we first introduce a set of quantities to formalize the previous observations. We model the I/O throughput (for both disk and network devices, and input and output data), sampled at regular time intervals, as a *discrete-time random process.* A random process is a real-value function $x(t, \omega)$, where ω is the outcome of a random experiment, and t is the independent variable representing the time. The random process can be viewed as a collection of deterministic time functions $x(t, \omega_i)$ (namely, *realizations* of the random process), corresponding to various executions ω_i of the experiment; alternatively, we can view the random process as a collection of random variables $X(t_i)$, corresponding to each time value t_i; if $t \in \mathbb{Z}$, it is a discrete-time random process. For a more rigorous and in-depth description of random processes, see [PP01] and [PS01].

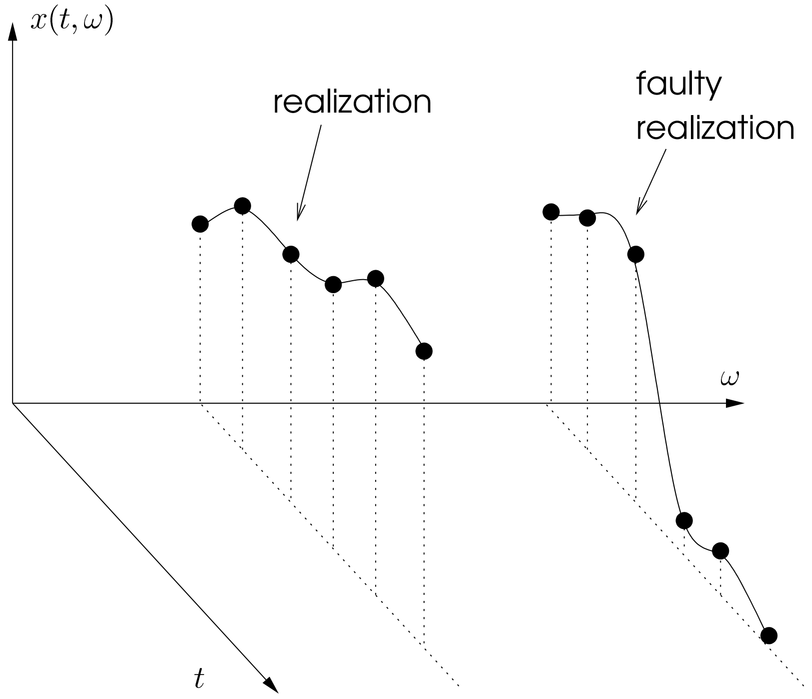


Figure 4.7: *A discrete-time random process.*

A process $x(t, \omega)$ is defined *stationary* if it meets the following conditions:

$$m_X(t) = E[X(t)] \text{ is independent of } t \tag{4.1}$$

$$R_X(t_i, t_j) = E[X(t_i)X(t_j)] \text{ depends only on } t_j - t_i \tag{4.2}$$

In addition to the statistical average $m_X(t)$ associated with each random variable $X(t)$ of the process $x(t, \omega)$, we can find the time average of each realization $x(t, \omega_i)$, defined as:

$$\langle x(t, \omega)\rangle_i = \lim_{N \to \infty} \frac{1}{N} \sum_{n=-N}^{+N} x(n, \omega_i) \tag{4.3}$$

In general, the real quantity $\langle x(t, \omega)\rangle_i$ is dependent on the realization ω_i considered, but it is independent of t. A stationary process is also said to be *ergodic* if, for each realization $x(t, \omega_i)$ and all functions $g(x)$:

$$\langle g(x(t, \omega))\rangle_i = E[g(X(t))] \tag{4.4}$$

The ergodicity is a very strong property implying that all time averages are equal to the statistical average $E[g(X(t))]$; it is supposed that the process is stationary, so the statistical average is a constant real value. Because the condition for ergodicity must hold for each $g(x)$, we have:

$$\langle x(t, \omega)\rangle_i = E[X(t)] = m_X \tag{4.5}$$

$$\langle (x(t, \omega) - m_X)^2\rangle_i = Var[X(t)] = \sigma_X^2 \tag{4.6}$$

A consequence of ergodicity is that we can measure various statistical averages, such as mean and variance, by looking at a single realization $x(t, \omega_i)$ of the process and evaluating its time averages, instead of considering a large number of realizations and averaging over them. If we assume the hyphotesis of stationarity and ergodicity for our processes representing the sampled throughput, we have that the mean value and variance of the throughput can be evaluated by time-averaging a large number of throughput samples;

we will describe later how hypotheses can be relaxed to make the approach more feasible.

We now apply our first consideration about observed behavior under failure (*there exists a difference in terms of sampled throughput from the expected behavior when a failure occurs*). The throughput during correct execution (the expected behavior) is represented by m_X; the actual throughput sampled in the time period t_i is the random variable $X(t_i)$. Therefore, we assume that *the probability that the throughput $X(t_i)$ is different than m_X, and in particular it is greater if a failure occurred*; mathematically, we state this hyphotesis by writing that there exists an $\overline{h} \in \mathbb{R}$ such that:

$$\mathcal{P}(|X(t_i) - m_X| \geqslant h \mid \mathit{Failure}) > \mathcal{P}(|X(t_i) - m_X| \geqslant h) \qquad \forall h > \overline{h} \quad (4.7)$$

For the sake of brevity, we refer to the event "$|X(t_i) - m_X| \geqslant h$" as "*ThDiff*" (Throughput Difference). Using the Bayes' Formula

$$\mathcal{P}(A|B)\mathcal{P}(B) = \mathcal{P}(B|A)\mathcal{P}(A) \quad (4.8)$$

we can write:

$$\mathcal{P}(\mathit{Failure} \mid \mathit{ThDiff}) = \frac{\mathcal{P}(\mathit{ThDiff} \mid \mathit{Failure})}{\mathcal{P}(\mathit{ThDiff})}\mathcal{P}(\mathit{Failure}) \quad (4.9)$$

Therefore, using the hyphotesis (4.7), we conclude:

$$\mathcal{P}(\mathit{Failure} \mid \mathit{ThDiff}) > \mathcal{P}(\mathit{Failure}) \quad (4.10)$$

From (4.10), it follows that the probability of failure occurrence increases if we experience a throughput difference; so, we consider the event "*ThDiff*" as a *failure symptom*, and in our algorithm we detect a failure by evaluating this event. Of course, the event does not imply that a failure actually occurred, but that there is an higher probability, and we rely on an external mechanism to establish *if* and *what* failure occurred; such decision can be made on the basis of the output of several, different failure detectors. Other possible conditions are:

- A failure occurred, but there was not a throughput difference from the expected behavior (*false negative*);

- A failure did not occur, although there was a throughput difference (*false positive*).

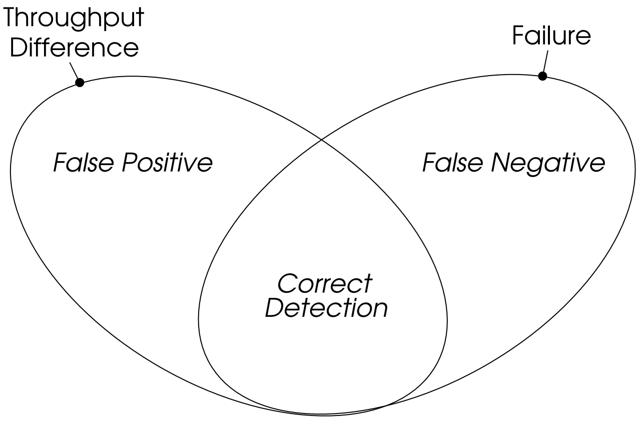


Figure 4.8: *Possible outcomes for the failure detector.*

The conditional probabilities $\mathcal{P}(\mathit{Failure} \mid \mathit{ThDiff})$ (correct detection), $\mathcal{P}(\mathit{Failure} \mid \mathit{not\ ThDiff})$ (false negative) and $\mathcal{P}(\mathit{ThDiff} \mid \mathit{not\ Failure})$ (false positive) depend on the choice of the h threeshold. Because we do not know the joint probability distribution of the random process, we also do not know the $\widehat{h}$ value that provides the desired probabilities of correct detection, false positive and false negative.

If we assume that the ratio $\frac{\mathcal{P}(\mathit{ThDiff} \mid \mathit{Failure})}{\mathcal{P}(\mathit{ThDiff})}$ increases with h (this follows by supposing that a very large throughput variation from the expected one is probably caused by a failure, and it is very unlikely produced by an healthy behavior), we can achieve an high probability of correct detection by increasing h; by doing so the probability of a false positive is reduced, but we also increase the probability of a false negative. Because we can quantify those probabilities only through experiments, the best trade-off is achieved by trials. It should be observed that a throughput difference can be produced by a variation of the workload (e.g. the creation of a process that makes several I/O requests), that also causes a change of m_X; for now, we assume the hyphotesis of stationarity (the mean m_X is constant for each time t_i), and we will address this problem later.

In order to choose a value of h that guarantees a fixed level of confidence, we consider the Chebyshev's inequality, which defines a lower bound for the probability of the *ThDiff* event:

$$\mathcal{P}(|X(t_i) - m_X| \geqslant h) \leqslant \frac{\sigma_X^2}{h^2} \tag{4.11}$$

By assigning $h = k\sigma_X$, the (4.11) can be written as:

$$\mathcal{P}(|X(t_i) - m_X| \geqslant k\sigma_X) \leqslant \frac{1}{k^2} \tag{4.12}$$

The expression (4.12) states that, for any probability distribution of $X(t_i)$, the probability of a throughput difference is no greater than $1/k^2$; for example, $X(t_i)$ falls outside of $[m_X - 2\sigma_X, m_X + 2\sigma_X]$ with a probability lower than 0.25, and outside of $[m_X - 3\sigma_X, m_X + 3\sigma_X]$ with a probability lower than 0.11. Therefore, we can reduce the probability of a throughput difference (and so the probability of a false positive) by choosing a proper value of k. The choice of $k\sigma_X$ as threeshold makes the confidence level of correct detection ($1/k^2$) independent from the probability distribution of $X(t_i)$ (which is unknown). The definition of the event "*ThDiff*" used by our algorithm is then

$$|X(t_i) - m_X| \geqslant k\sigma_X \tag{4.13}$$

At the time t_i, the throughput is sampled, and we refer to the actual value assumed by the random variable $X(t_i)$ as $\overline{x}_i$. In order to check if the event (4.13) occurred, we have to estimate the mean and the standard deviation of $X(t_i)$, represented by m_X and σ_X. Until now, m_X and σ_X were constant quantities for all t_i, because we have made the assumption of process' stationarity; moreover, if the ergodicity hyphotesis holds, m_X and σ_X at t_i can be evaluated by using the time averages (4.5) and (4.6).

Nevertheless, exact calculation of those averages is not feasible because it requires an infinite number of samples; moreover, the assumptions of stationarity and ergodicity of the random processes are not guaranteed. We can obtain at least an estimation of mean and standard deviation, if considerations made in the previous section are verified by experiments on the system in hand: we are going to show that, if these considerations hold, the previous hypotheses on the random processes are not needed to evaluate means and standard deviations. By applying our last consideration (*the expected behavior can be approximated by the observed behavior under failure, if it is evaluated in a time period enough long.*), we estimate statistical averages of $X(t_i)$ by using time averages of a (finite) number of samples in different instants:

$$m_X = \frac{1}{N} \sum_{k=i-N}^{i-1} \overline{x}_k \tag{4.14}$$

$$\sigma_X = \sqrt{\frac{1}{N-1} \sum_{k=i-N}^{i-1} (\overline{x}_k - m_X)^2} \tag{4.15}$$

When computing time averages, we have taken into account the other two considerations we made: the first (*Even when there is a failure, the expected behavior can be approximated by the average throughput in a time period including non-faulty behavior*) imposes the use of samples preceding the occurrence of a failure, because samples following a failure are not representative of the expected (correct) behavior; the second (*the expected behavior can be approximated by averaging the observed behavior in a longer period, in which the workload does not vary noticeably*) limits the number N of previous samples that can be used, because the samples $X(t_{i-N}) \dots X(t_{i-1})$ should have statistical averages close to the ones of $X(t_i)$, in order to make an accurate estimation of mean and standard deviation.

Because the statistics of the probability distribution of $X(t_i)$ can differ for each t_i, the averages (4.14) and (4.15) should be re-computed for each i. However, because the workload imposed to the system can suddenly vary, the averages estimated using previous samples can significantly differ from the statistical averages in t_i, leading to a throughput difference (false positive); such problem could be limited only by a suitable choice of the sampling period and the number N of samples. In the next section, we discuss the problem of choosing the parameters characterizing the detection algorithm in order to meet the required preconditions.

4.4 Tuning of the algorithm

The described algorithm just allows the detection of failure modes which are related to performances changes, in respect to the performances expected for a correct behavior of the system. This technique requires that the throughput of device drivers (the number of I/O operations in the time unit) is periodically sampled, and that the current sample is compared to the previous ones through the relationship (4.13); a warning should be produced if the computed value exceeds a threeshold. Therefore, the following parameters have to be defined:

- the period of time T between consecutive samples;
- the number N of previous samples used to estimate the mean and the standard deviation;
- the number k to which the standard deviation is multiplied to obtain the threeshold.

The choice of the optimal values for these 3 parameters maximize the probability of correct detection, and minimize the probabilities of a false positive or a false negative. The difficulty in making a good choice is due to the unavailability of a complete statistical description of the random processes. In the following we suggest several guidelines to help in the choice, allowing to base the detection algorithm on the information gathered from preliminary experiments; the approach does not guarantee an optimal choice of the parameters, but it leads to a good trade-off, that we will evaluate in the next chapter.

In order to make a proper choice during the pre-operational phase of the system, we base our procedure on the analysis of a representative workload. Experiments are executed to trace the occurrence of I/O events for the considered workload; the events of interests are:

- reception of a packet, from the network device driver;
- transmission of a packet, to the network device driver;

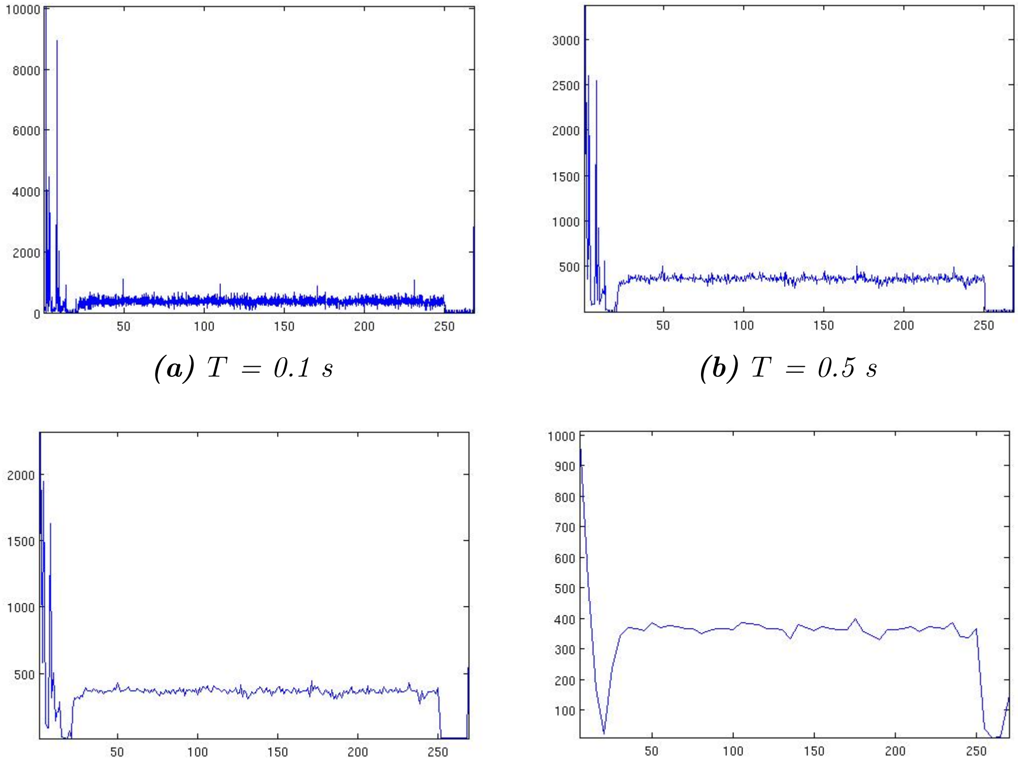
- read of a block, from the disk device driver;
- write of a block, to the disk device driver.

Once all events of interests during the execution have been traced, the sampling period T is chosen. We consider a finite set of possible values, and for each T we plot the number of events counted by each sample. The considered values of T should be:

- much greater than the mean interarrival time between consecutive packets or blocks, in order to avoid the presence of a large number of samples equal to zero;
- less than the period of time between significant variations of the sampled throughput.

In our example, we evaluate the following candidate values for T: 0.1, 0.5, 1 and 5 seconds (all greater than the 99th percentile of the interarrival time distribution). The value of T should be chosen such that the plot of the samples accurately describes variations of the throughput during the execution, but removing short fluctuations due to secondary factors (e.g. $T = 0.1$ s), and avoiding a too coarse representation in which variations between consecutive samples are too abrupt (e.g. $T = 5$ s). By the comparison of the different plots obtained, we choose $T = 1$ s as the best representative of the throughput's trends; higher values has to be rejected because of the presence of too abrupt differences between consecutive samples. The problem of the choice of T is similar to the event tupling, in which correlated error log entries should be grouped by their timestamp and type [BS96].

Once the value of T has been fixed, we have to choose the values for the N and k parameters. The former should be such that the throughput is roughly stable in N samples: there are not too much significant throughput variations in a period of $N \cdot T$ seconds, and the samples allow for a good approximation of the expected throughput m_X. For each suitable value of N, we have to identify the value of k such that random variations from the current mean are discriminated from those caused by a failure. Therefore, we plot the maximum value k_i that, for each sample i, verifies the (4.13):



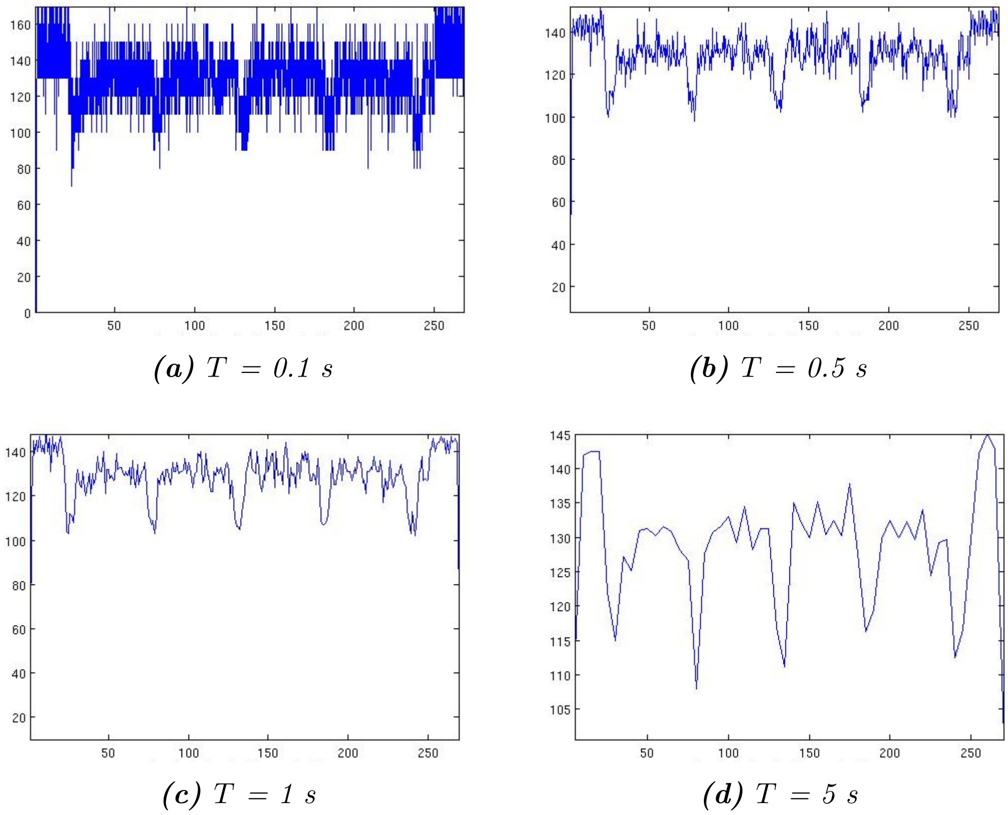
(a) *T = 0.1 s*

(b) *T = 0.5 s*

(c) *T = 1 s*

(d) *T = 5 s*

Figure 4.9: *Plots of throughput samples for packet transmission.*



(a) *T = 0.1 s*

(b) *T = 0.5 s*

(c) *T = 1 s*

(d) *T = 5 s*

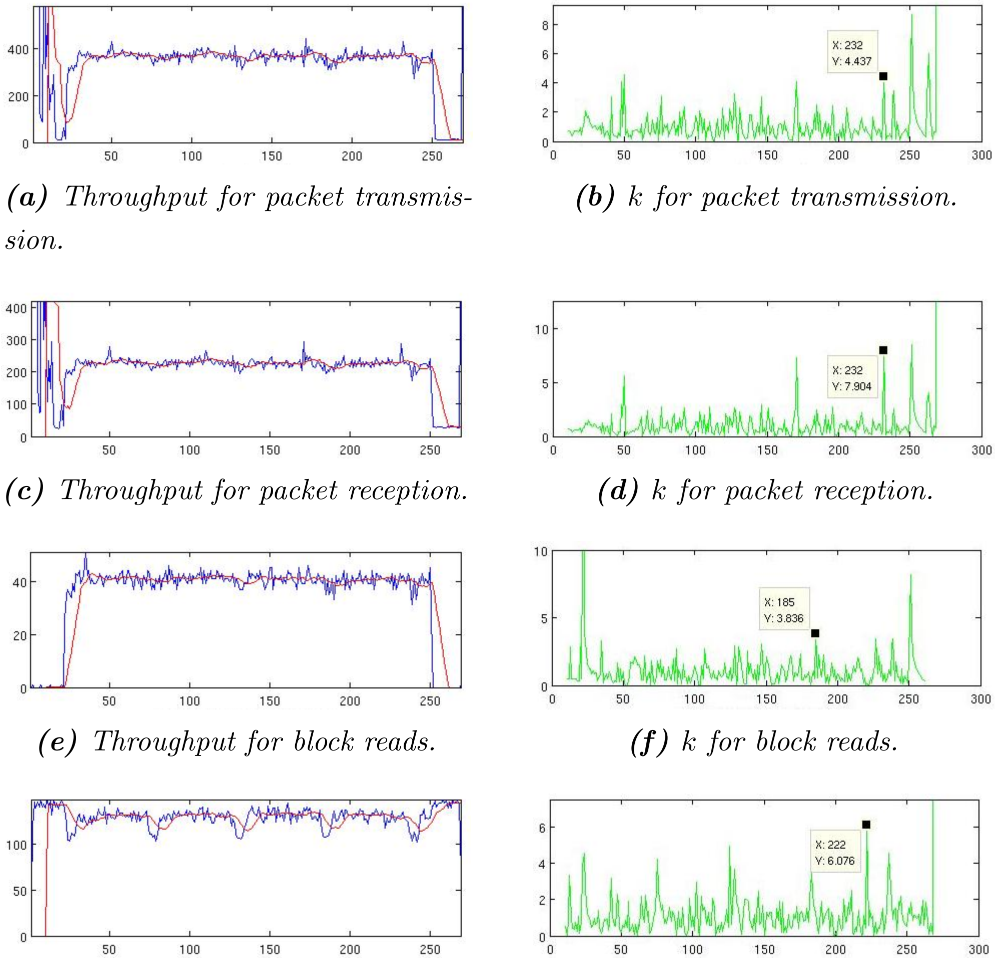
Figure 4.10: *Plots of throughput samples for block writes.*

because we know *a priori* that a failure did not occur in the period t_i, we must choose a value of k greater than $k_{\max} = \max_i k_i$ (so that a false positive is not produced by the algorithm); vice versa, k should be less than k_i if we know there is a failure in t_i (e.g. we injected a fault that was activated, and a failure should be detected).

To explain the described procedure by examples, we show the plots (figures 4.11, 4.12 and 4.13) of k_i for each sample i and for different values of N (10, 30, and 50), during the execution of a workload without injection. The plots also include observed throughput before and after workload execution (the workload actually executed for $t \in [25, 250]$): when the workload is started ($t \backsimeq 25$) and stopped ($t \backsimeq 250$), there were values of k_i greater than the values during the execution, because there are an abrupt rise and fall before and after the execution, respectively; we must choose N and k such that those samples are recognized as symptomatic (that is, much greater than the estimated mean), because the workload alternates between working and non-working conditions (we expect a similar behavior when a failure occurs). We have to choose the values of N and k for each monitored event (those values can differ between events); k should be greater than all k_i during correct workload execution, and less than k_i at the beginning and at the end of the execution (or, in general, when it is known that a failure occurred in t_i).

In figure 4.11, we observe that the choice $N = 10$ does not allow to discriminate between the values of k_i during correct execution and k_i during throughput variations at the beginning and at the end: for packet reception plot (figure 4.11d), the maximum k_i during workload execution (7.904) is near to the value of the peak at the end; also, in block writes plot (figure 4.11h) there are significant peaks of k_i, so we discard the $N = 10$ value. In general, the values chosen for T and N should not produce too high values of k_i, because this fact represents an high variance of sampled throughput values, that makes difficult to distinguish failure symptoms.

For $N = 30$ (figure 4.12), we obtained a better discrimination between values of k_i: a threeshold of $k = 5$ or $k = 6$ prevents false positives (for each t_i, $|X(t_i) - m_X|$ is less than $5 \cdot \sigma_X$), except for packet reception (figure 4.12d); for $N = 50$, also packet reception peaks are correctly discriminated (figure 4.13d). Nevertheless, such a great value for N requires that the workload does not change significantly during a period of 50 seconds (remember that the stationarity hypothesis is needed to estimate the mean value using previous values); in our example this condition is true, but a more conservative choice for N is suggested if we do not have an high confidence of this assumption. In order to gain more knowledge about the workload of the system, and to obtain an higher degree of confidence, the previous analysis should be repeated for different execution traces.



(a) Throughput for packet transmission.

(b) k for packet transmission.

(c) Throughput for packet reception.

(d) k for packet reception.

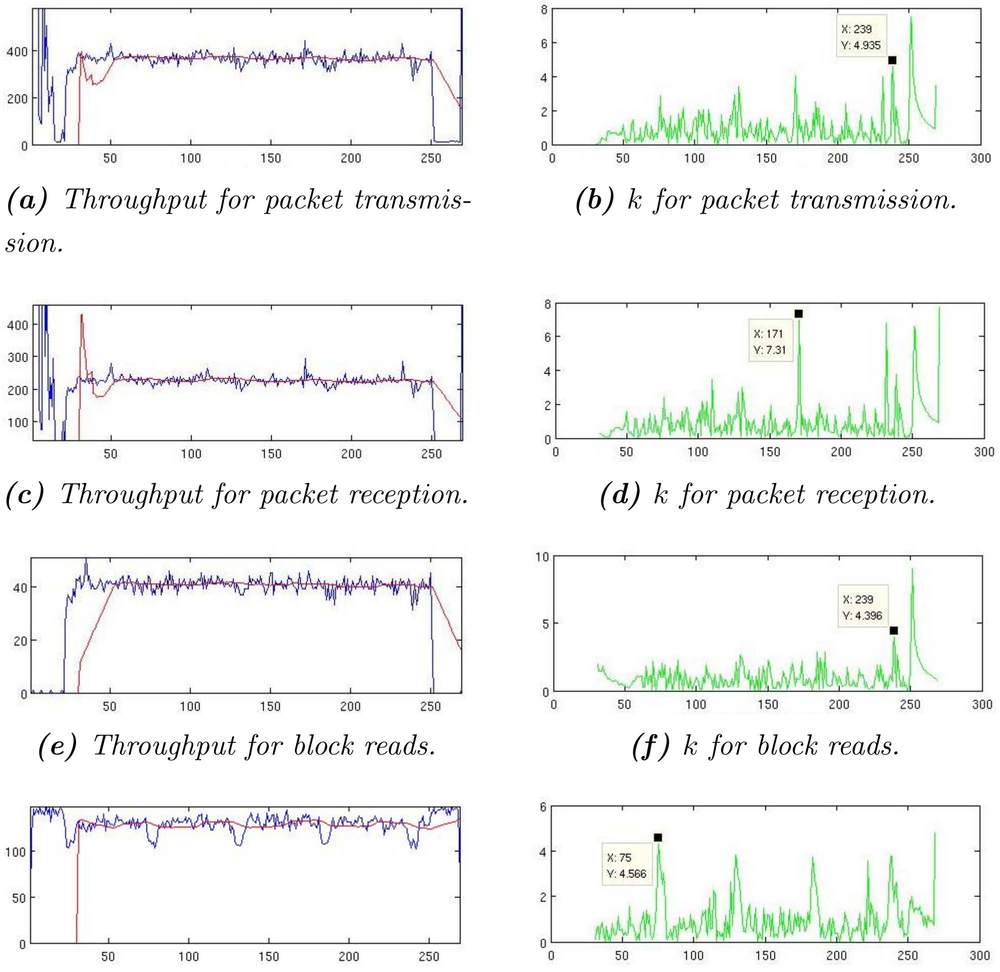
(e) Throughput for block reads.

(f) k for block reads.

(g) Throughput for block writes.

(h) k for block writes.

Figure 4.11: *Plots of throughput (blue), estimated mean (red) and maximum instant values of k for detection (green), with N = 10.*



(a) *Throughput for packet transmission.*

(b) *k for packet transmission.*

(c) *Throughput for packet reception.*

(d) *k for packet reception.*

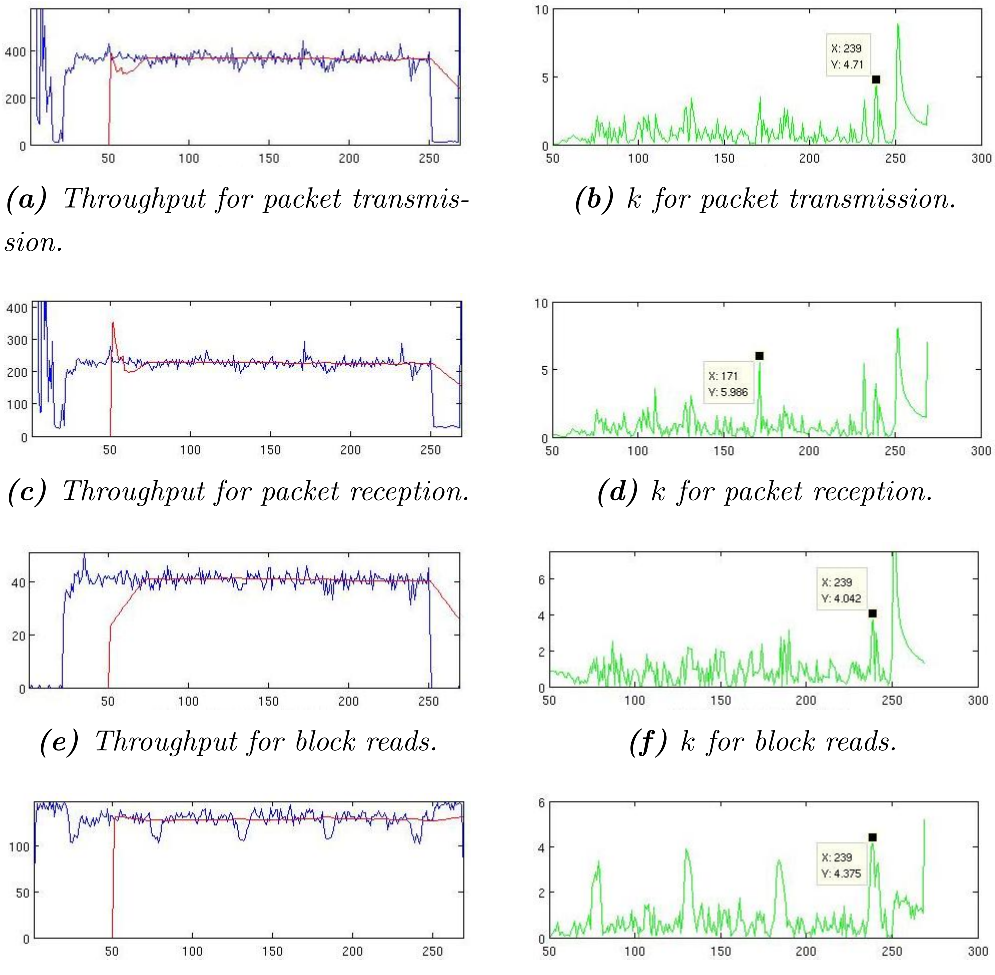
(e) *Throughput for block reads.*

(f) *k for block reads.*

(g) *Throughput for block writes.*

(h) *k for block writes.*

Figure 4.12: *Plots of throughput (blue), estimated mean (red) and maximum instant values of k for detection (green), with N = 30.*



(a) *Throughput for packet transmission.*

(b) *k for packet transmission.*

(c) *Throughput for packet reception.*

(d) *k for packet reception.*

(e) *Throughput for block reads.*

(f) *k for block reads.*

(g) *Throughput for block writes.*

(h) *k for block writes.*

Figure 4.13: *Plots of throughput (blue), estimated mean (red) and maximum instant values of* k *for detection (green), with* $N = 50$.

4.5 Conclusions

In this chapter, a technique for failure detection on complex systems was introduced. The technique allows for the detection of such failure modes (of the system as-a-whole) leading to performance variations (e.g. crashes, hangs), and it is based on performance measurements at the operating system level: in our work, we focused on the sampling of the throughput of device drivers operations, that is packets exchanged by network devices and blocks read and written by disk devices. The analysis of faulty and non-faulty executions of a same workload showed that:

- The occurrence of a failure affects the considered device drivers measurements (compared to the cases of correct execution);
- The measurements during non-faulty execution periods can be used to estimate the expected throughput at a given time.

From those observations, we derived an algorithm that, using previous throughput measurements, checks for significant variations from the expected throughput at a given time; if it is the case, a failure is probably occurred and a warning should be produced. Because the throughput is a quantity subject to variations during nominal execution, a variation is considered as a failure symptom if it is much more sharp than the previous ones, using an estimate of the standard deviation of the measurements. The threeshold applied by the algorithm, and other parameters such as the sampling period, are tuned by the observation of the execution of a representative workload, in order to achieve the desired trade-off between correct detection of anomalous variations and wrong detections, by varying the degree to which such an event is considered as a symptom.

CHAPTER 5

Experimental evaluation of the detection technique

> An algorithm must be seen to be believed.
>
> Donald Knuth

5.1 Introduction

In this chapter, the detection technique introduced in the previous chapter is evaluated using a testbed representative of mission critical applications. The CARDAMOM middleware, briefly described in the next section, is adopted to deploy a distributed fault-tolerant application; faults will be injected at the operating system and at the application level, and the latency and coverage of the failure detector will be measured under this scenario.

5.2 The CARDAMOM middleware

CARDAMOM is an opensource, CORBA-based middleware for safety and mission critical applications [1]; it supports development of applications based on the Distributed Object Model (DOM) and the Component Model (CM).

CARDAMOM implements (and makes further additions to) the services prescribed by the CORBA standard (see [HV99] and [RSCS02]), focusing on scalable and safety-critical applications (figure 5.1), such as:

System Management Service Dynamic configuration and monitoring of all nodes, processes and components in the system;

Fault Tolerance Service Transparent replication of the application, including mechanisms for failure detection, request retry and redirection, and state consistency between replicas.

Event Service Asynchronous, publish-subscribe communication.

Load Balancing Service Transparent redirection of requests toward a pool of servers, in order to balance the load between them.

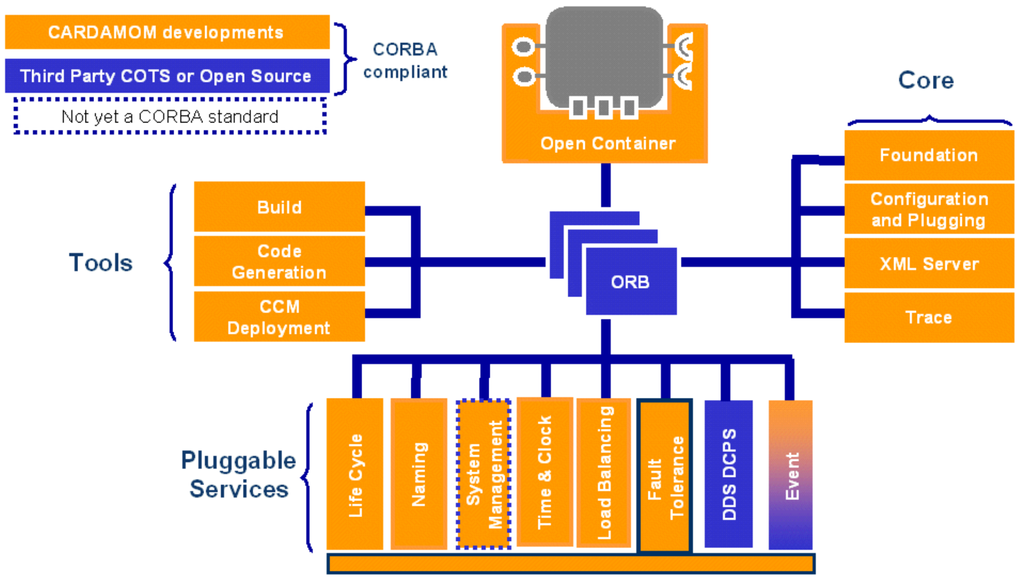


Figure 5.1: *Services provided by the CARDAMOM CORBA-based middleware for mission and safety critical applications.*

The System Management can be used to deploy applications to several nodes, to initialize, start and stop system's elements in the desired order (allowing the deployment of complex scenarios), to collect informations about the state of the applications, and to send/receive notifications to/from the system. If a process' life cycle is completely handled by the System Management, it is a *managed process*. The System Management Service consists of the following processes:

Platform Supervision Server An unique process for supervising all applications running on the system; it offers IDL interfaces to developers and to applications for the management of the system.

Platform Daemons Processes running on each host on the system; they control the Application Agents running on their node.

Application Agent It starts, stops and monitors processes of a particular application, running on its node.

The System Management defines an interface to allow applications to send their state, errors notifications and other information to the Supervision Server; moreover, a servant (the implementation of a CORBA object) of a managed process has to implement the following *behavioural methods* that are invoked by the platform at the initialization, start and stop of the process:

on_initialise invoked when the process is initialized;

on_next_step the process may require several phases for its initialization, which are defined in this method;

on_run invoked when the process begins its execution (running state);

on_stop invoked when the process is stopped.

These methods are called through the *callback* mechanism: a reference to the object is passed to the platform when it is first instantiated, and the methods are asynchronously invoked during process execution.

The Fault Tolerance (FT) CARDAMOM Service is based on the FT CORBA Specification. The latter prescribes that a CORBA object should be replicated (i.e. more than one CORBA object and servant should be instantiated in order to provide the same functionality), to ensure that requests will be processed even if a subset of replicas is failed. The full set of objects is called *object group*, which can be referenced through a single identifier, namely IOGR (Interoperable Object Group Reference); the client is unaware of object replication, and client's requests made to the IOGR are transparently forwarded to the objects of the group. In general, the object replication matches with the replication of processes and physical nodes. The middleware is responsible for the detection of objects' failures, which is followed by failure notifications and recovery interventions.

The state of the object group is shared between object replicas. Several replication schemes are described by FT CORBA:

Stateless No state is assumed by the objects; requests are not influenced by the past ones.

Cold Passive Requests (and the state) are only managed by a single (*primary*) object in the group; when the primary object fails, another object (*backup*) retrieves the last snapshot of the state (*checkpoint*) from the primary server.

Warm Passive As cold passive replication; checkpoints are periodically sent to the replicas.

Active Requests are independently executed by each object of the group; results are collected by a single server (*gateway*), which returns any of the replies to the client. In order to ensure the consistency between replicas (they should execute the same requests in the same order), a Reliable Multicast protocol has to be used.

Active with voting As in active replication; objects' replies are compared, and replies different from the majority are discarded by the gateway.

The replication scheme adopted by the FT Service is the Warm Passive Replication. Developers have to implement the state checkpointing and consistency, and manage the object group's memberships; CARDAMOM offers dedicated facilities for these tasks, such as the State Transfer Framework (for transactional state synchronization).

In CARDAMOM, replicated objects (primary and backups) are contained in separated processes, namely *FT Locations*. These processes are monitored by the FT Manager (figure 5.2), which is in charge of detecting failures, by periodic polling of daemons on different nodes (*Simple Monitoring*), which, in turn, query local processes; if an answer is not returned within a timeout, a process is considered faulty. The FT Manager, in turn, can be replicated to improve system's reliability. When a failure is discovered, notifications are sent to the Replication Manager and to the Platform Supervision: the former manages object groups, by remotely deactivating failed processes and activating a backup replica; the latter may schedule a general application reconfiguration, depending on the applicative scenarios defined by developers.

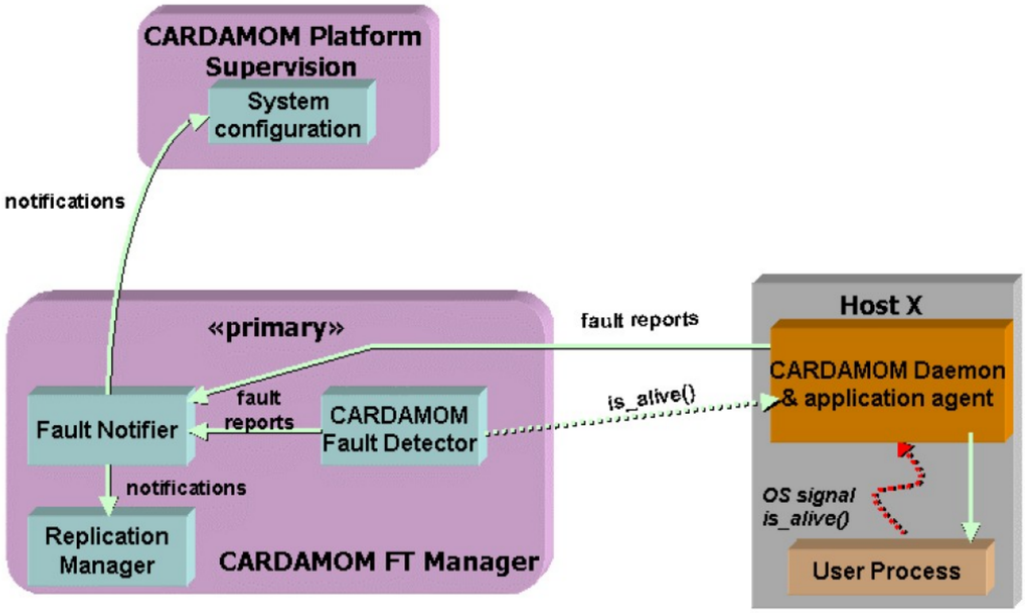


Figure 5.2: *Architecture of the Fault Tolerance Service. The FT Manager periodically polls platform daemons to inspect processes' status. If a failure is detected, the Replication Manager activated a backup replica of the object group.*

5.3 Experiments setup

The workload considered in this chapter is a distributed application, in which a server replies to requests made by several concurrent clients. This workload is representative of applications dedicated to remote data collection and storage; in this scenario:

- clients send data to the server by remote invocations of a method of a distributed CORBA object;
- data is contained in a complex, nested structure, which requires a non negligible overhead for un-serialization by the server;
- the server writes to its local disk several logging information and the contents of a request; it also returns a integer value read from a local file;
- at each request, the server increases a local counter which represents its state information.

The deployed workload makes use of the Fault Tolerance Service provided by CARDAMOM; the server is replicated two times (one primary and two backup servers), and the FT Manager configures the platform to redirect requests to a replicated server in the case of failure of the primary one. The server state (the value of its internal counter) is synchronized with replicas, which do not service requests until a server's failure; when a replica is activated, it can seamlessly keep running the distributed application without apparent service interruption or failures. The objects and processes composing the workload are distributed in 8 identical machines:

- 3 nodes execute, respectively, the primary server and 2 backup replicas;
- 3 nodes execute client processes;
- 1 node executes the FT Manager;
- 1 node executes the Supervision Server.

The hardware and software configuration of a node is shown in table 5.1. Each node mounts an NFS remote root partition, and several other NFS partitions for programs, shared libraries and home directories; a node is also equipped with a local SCSI disk, in which are stored logs from applications and the middleware, and files read and written by the workload.

Table 5.1: *Hardware and software configuration of the distributed application nodes (HP ProLiant DL380 G3 Server).*

CPU	Intel Xeon^{TM} 2.80GHz (dual processor)
RAM	3.5 GB
Disk controller	Compaq Smart Array 5i/532
Hard disk	Wide Ultra3 SCSI Hard Drive, 36 GB, 160 MB/sec (10000 RPM)
Network card	Broadcom NC7781 Gigabit Ethernet
Linux kernel version	2.6.21
Linux distribution	Red Hat Enterprise Linux AS release 4 (Nahant Update 2)
CARDAMOM version	3.1
SystemTap	snapshot 20070721

The primary server is the node considered for fault injection and failure detection experiments. The clients generate a fixed amount of requests (1500 requests/client, 4500 total requests), during which a fault is injected (about half a minute since the start time of the execution). The time between consecutive requests ranges from 10 ms to 50 ms, and it is progressively increased and decreased during an experiment, in order to evaluate the detector under load fluctuations.

The measures which we will get from the experiments are the coverage and latency of both the operating system and the described failure detector, as defined in the section 3.3.1. Our purpose is to compare those measures, to show that the failure detector can be used by the system to improve its detection capabilities. As prescribed by the dependability benchmark, the considered faultload includes injections at the device drivers' interface; the same implementation of section 3.3.2, by dynamic probing of target functions. The experiments in this chapter also share the considerations of the previous dependability benchmark, such as the possible outcomes, the statistical interpretation of the results, and the way the measures are estimated from the experiments.

Moreover, software faults in the user application level are injected, in order to evaluate the coverage and latency of the detector in respect to system-level failures induced by applications; for application faults, the failure detector's coverage and latency are compared with the ones of the FT Manager detection mechanism. As explained in section 5.5, the injected software faults are chosen such that the failures produced by them are representative of the real ones.

The measured throughput for this workload was also used in the previous chapter to explain the tuning of the algorithm (§ 4.4). Therefore, in our assessment experiments, the same parameters for the failure detector described there will be used:

- A sample period of $T = 1$ s;
- A number of previous samples $N = 30$;
- A threeshold $k = 6$ for both packet transmission and reception;
- A threeshold $k = 5$ for both disk reads and writes;

The I/O throughput, as seen by the failure detector, is shown in figure 4.12. When a failure is detected, a log entry is produced by the detector, which is composed by the following fields:

- A timestamp (in nanoseconds);
- The event which experienced the throughput difference (network transmit/receive, disk read/write);
- The device (network interface or disk device) which the event refers to;
- If the throughput difference is a throughput peak (the sampled throughput exceed the estimated mean value) or a throughput loss (the mean exceed the sampled throughput);

- The value of the current throughput sample;
- The mean estimated through previous throughput samples;
- The variance estimated through previous throughput samples;
- The value of k used for the threeshold.

The detector was implemented in the SystemTap scripting language (listing A.7), and it executed as part of the workload in kernel space; because of potential disk driver failures, the detector log entries were also redirected to the system console (`/dev/console`). Log files were also produced by the CARDAMOM middleware and by the client and server processes; they include the outcome of each disk operation by the server, the server state and returned values, the serviced requests, failures detected and processed by the FT Manager, and the timestamp of each entry.

5.4 Operating system faultload

5.4.1 Description

The first test set used to evaluate the detection algorithm is based on a faultload at the operating system level (figure 1.5). The same dependability benchmark defined in section 3.3.1 was used, with the workload configuration described in section 5.3. Again, the activation of software faults was emulated by error injection, through parameter corruption of functions' invocations at the interface between device drivers and the rest of the kernel. The injected errors should not be confused with SWIFI experiments described in section 4.2; the latter are representative of hardware-induced faults, but not of real software defects in the source code.

In order to improve the effectiveness of tests, i.e. to reduce the number of experiments and to increase the number of robustness failures found, still producing a similar estimation of coverage and latency, the results of experiments described in section 3.3.3 were closely analyzed. An analysis of the outcome of each test revealed that:

- The flipping of a subset of bits produced twice the number of system-level failures than other bits;
- The most of system-level failures was caused by injection at a small subset of function parameters.

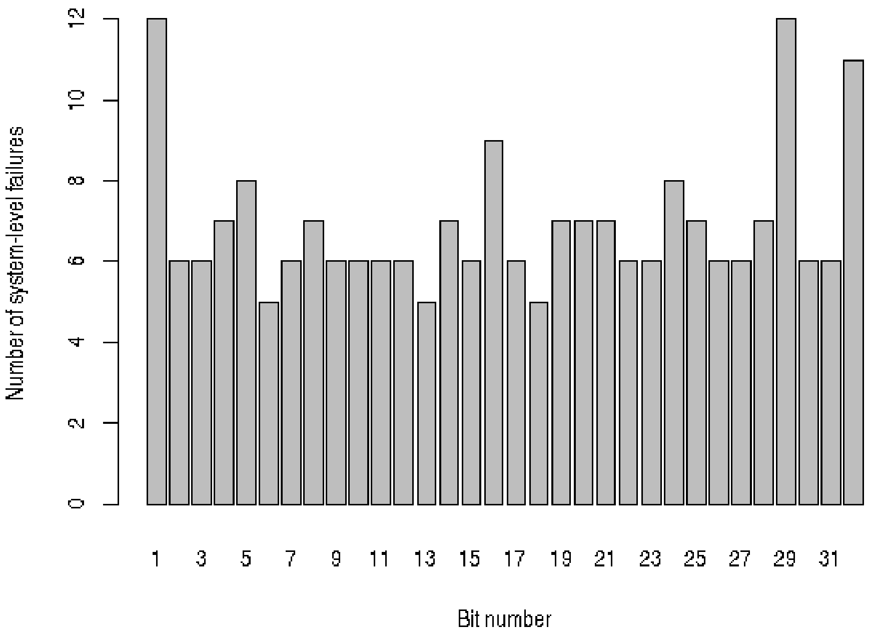


Figure 5.3: *Distribution of system-level failures across the bits in which a bit-flip is injected.*

As showed in figure 5.3, the 1st, 29th, and 32th bit produced a greater amount of system-level failures. Moreover, there were a subset of parameters in which many bit flips caused a system-level failure (namely, vulnerable parameters), and a little number of failures was observed for the rest of function parameters (namely, robust parameters); for the tg3 driver, there were 7 out of 21 vulnerable parameters, and 1 out of 5 for the scsi driver (figure 5.4). Therefore, we can improve the efficiency by focusing the test selection on the most vulnerable bits; because outcomes are grouped by parameters (bit-flips on vulnerable parameters lead often to system-level failures, and bit-flips on robust parameters do not significantly affect the experiment), and some bits

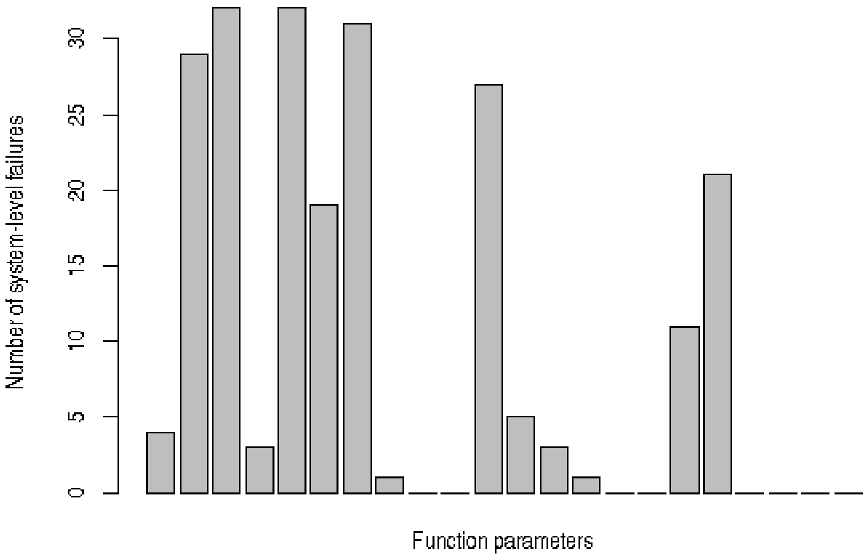


Figure 5.4: *Distribution of system-level failures across the params in which a bit-flip is injected.*

are more vulnerable than other (in term of produced failures), thus a great number of system-level failures can be observed if we limit bit-flips only to the most vulnerable bits, still obtaining representative results (because a failure after injection at a vulnerable parameter is representative of all other potential failures produced through that parameter). In our experiments, we will inject bit flips only on bits 1, 29 and 32, instead of full 32 bit flips for each parameters, thus reducing of a factor of 11 the number of experiments.

The device drivers subject to error injection were the scsi disk driver (*cciss*) and the network card driver (*tg3*). The target functions were selected by finding the most called functions of the kernel called by the drivers, and the most called functions of the drivers called by the kernel. A set of 48 experiments for the scsi driver (16 parameters of 11 functions), and a set of 24 experiments for the network driver (8 parameters of 7 functions) were selected.

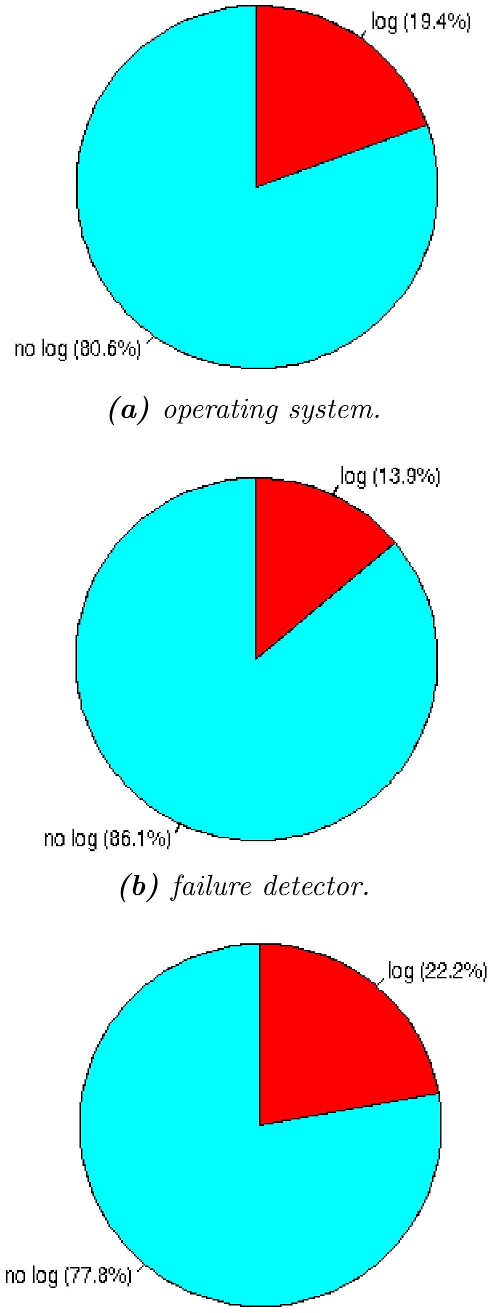
5.4.2 Results

In figures 5.5a and 5.5b, it is shown the number of experiments in which a log entry was produced by the system and by the failure detector, in respect to the total amount of experiments, for the considered System Under Benchmark; the ratio represents the coverage in respect to both scsi and network drivers' failures. These results allow the comparison of the percentages of the injected driver's failures which were detected by the proposed algorithm and by the system's logging mechanisms. In figure 5.5c, the coverage obtained by merging logs from the detector and the system is indicated.

The figures 5.6a, 5.6b, and 5.6c show the coverage in respect of system-level failures, i.e. the subset of injections not tolerated by the system. The most of non-logged system-level failures were observed for the network driver, in which a not valid pointer is immediately dereferenced after injection, leading to the complete interruption of kernel execution; this is the same cause of low system's coverage for network driver's failures of section 3.3.3.

From these results, we observe that the failure detector did not increase significantly the coverage in respect to tolerated drivers' failures, because the overall coverage is close to the one of system's logs. Instead, the improvement of the coverage in respect of system-level failures (the most severe ones) is greater. This is due to the fact that when a failure is not tolerated by the operating system, it affected the whole system (including the workload), leading to performance variations; because the failure detection algorithm assumes that a failure will produce sudden variations of the system's behavior in respect to I/O operations, it can not detect a failure which not modifies significantly the system's behavior (this is the case of a driver's failure which is tolerated by the rest of the kernel). Because there exist cases in which the algorithm detected failures not logged by the system, the failure detector can be included in system's logging facilities, in order to improve the degree of detection of critical failures.

In table 5.2, it is shown the mean latency for the failure detector and system's logs, for both scsi and network drivers' failures. In all experiments, a low latency (about 1 second) was observed; the mean is evaluated from those cases in which it was possible to quantify the detector logging delay;

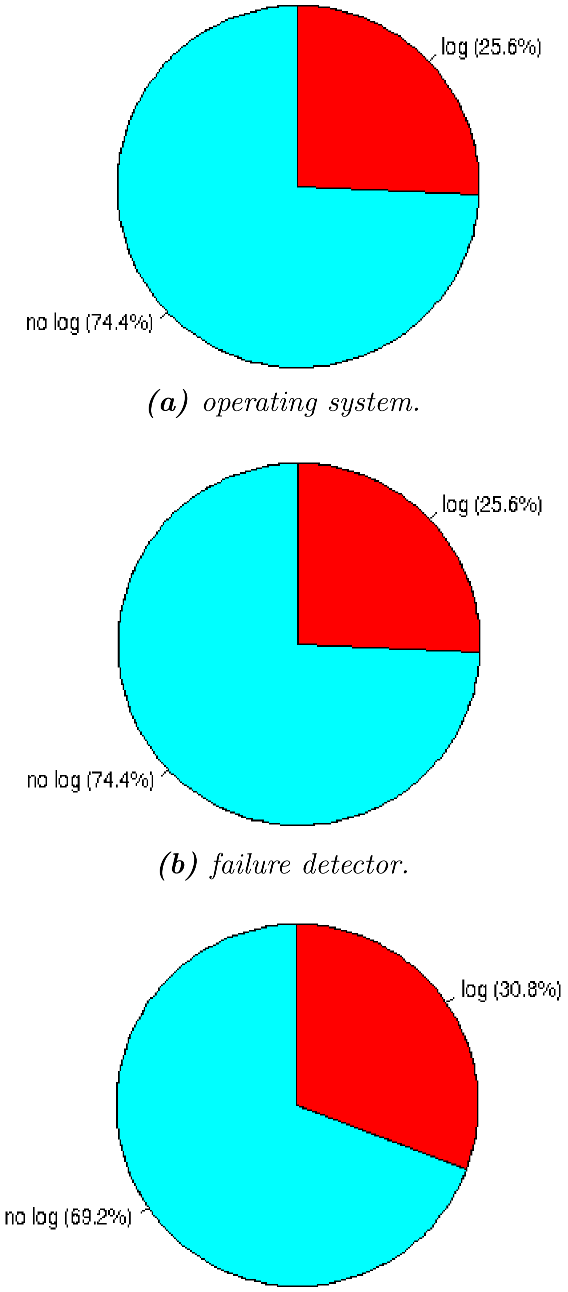


(a) *operating system.*

(b) *failure detector.*

(c) *merged logs.*

Figure 5.5: *Log coverage for injections at the operating system level.*



(a) *operating system.*

(b) *failure detector.*

(c) *merged logs.*

Figure 5.6: *Log coverage of system-level failures for injections at the operating system level.*

in other cases, we know that the actual detector latency was lower than 1 second, although it was not possible to quantitatively measure it. The failure detector did show an higher latency, due to the higher sample period ($T = 1$ s) and the algorithm overhead, while the standard log mechanisms detected failures by immediate checks on the system's internal state (e.g. correctness checks of the parameters passed to a function). Nevertheless, the detector's latency was significantly lower than high latency cases observed in experiments of section 3.3.3.

Table 5.2: *Mean detection latencies for failures induced at the operating system level. The failure detector samples the I/O throughput with a sampling period of 1 second.*

	Failure Detector	Operating System
Mean latency	0.403760 s	0.036916 s

5.5 Application faultload

5.5.1 Description

In order to assess the quality of the proposed failure detector, we also executed several fault injection experiments at the application level. Because the operating system is not affected by these injections (the system call interface was proved to be robust), we do expect that the operating system's facilities will not produce log entries related to experiments, therefore the detection coverage and latency evaluated by these experiments are referred to system-level failures, i.e. failures that affect the application level and thus the behavior the system as-a-whole.

The faults to be injected in the server process are based on the failure modes described (and expected to be tolerated) in the specifications of the FT CARDAMOM service; this service should replace a primary copy of an object if its process crashes, or it is under suspicion to be faulty. We can identify two classes of failures:

- Failures implying the arrest of a servant's process (*server failures*);
- Failures in which application's behavioral methods (callbacks) execution exceeds a predefined amount of time (*timing failures*).

The considered failure classes include process crashes and hangs; in particular, we assume that these failures are caused by software faults in the servant (the implementation of the CORBA object) or in callback methods. Process hangs can be further differentiated in *active hangs* (the process still executes and makes use of system's resources, although it is indefinitely waiting) and *passive hangs* (the process is blocked and not executed, until a logical resource is available); an hang can be caused by an erroneous application state (e.g. it is in an infinite loop), by an undisciplined access to shared resources (e.g. a deadlock), or by the progressive consumption of physical or logical resources; these conditions more probably occur for concurrent, real-time, heavy-loaded systems, and attention should be paid to cope with them in critical applications. Instead, a crash can be caused by the voluntary return of the process (e.g. after the catch of an exception), or a signal sent by the operating system (e.g. sigkill). Process crashes and hangs were also observed when errors were injected in the kernel (workload failures).

In order to induce the described failures, the source code of the workload application was mutated; after each mutation, the primary server's executable is substituted with the mutated one. The mutations used for the robustness test, implemented in [Cin07], are listed in table 5.3.

Table 5.3: *Source code mutations for software fault injection at the user application level.*

Identifier	**Type**	**Location**	**Mutation**
ServerFailure1	Crash	`on_run` callback	exit of the process after an amount of time
ServerFailure2	Crash	servant object	exit of the process at a predefined request

Continued on next page...

Table 5.3 – Continued

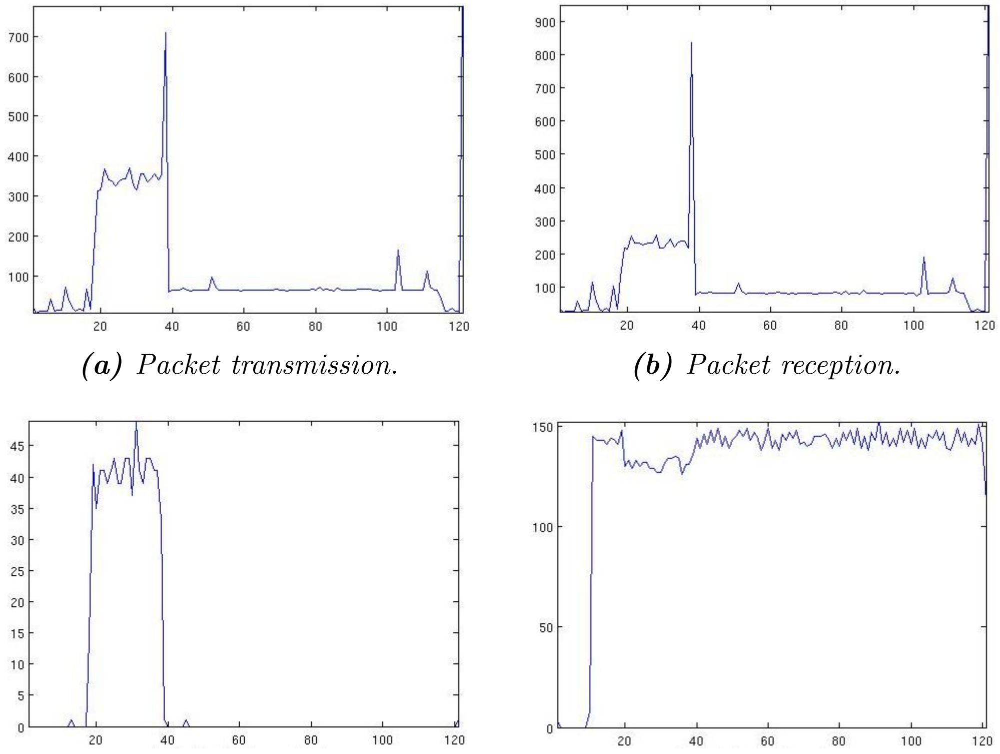
Identifier	Type	Location	Mutation
ServerFailure3	Crash	`on_run` callback	killing of the process after dereferencing a null pointer
ServerFailure4	Hang	servant object	infinite empty loop at a predefined request
ServerFailure5	Hang	servant object	infinite CPU and I/O bound loop at a predefined request
ServerFailure6	Hang	servant object	passive wait for an IPC semaphore at a predefined request
TimingFailure1	Hang	`on_run` callback	infinite empty loop
TimingFailure2	Hang	`on_run` callback	infinite CPU an I/O bound loop
TimingFailure3	Hang	`on_run` callback	passive wait for an IPC semaphore
HangOverloading1	Hang	servant object	progressive allocation of threads passively waiting for an IPC semaphore
HangOverloading2	Hang	servant object	progressive allocation of threads executing an infinite CPU and I/O bound loop
ServerFailureState1	Crash	`set_state` method	active finite wait and process exit at a predefined request
ServerFailureState2	Crash	`set_state` method	process exit at a predefined request
ServerFailureState3	Hang	`set_state` method	infinite empty loop at a predefined request

The considered failure modes are induced by injecting software faults of the Algorithm and Timimg/Serialization ODC classes (table 1.1); the faults were selected to ensure the immediate manifestation of considered failures: the execution flow of the server is altered to cause instantaneous process exit (by the `exit()` library call, or forcing a kill by dereferencing a null pointer), or a process stall, both passive (by requesting an IPC semaphore never freed) and active (by executing an infinite loop). Moreover, in order to reproduce stressing conditions of limited resource availability (*overloading*), which are recognized as a cause of failures, several threads (using the POSIX pthread library) were allocated at each request, thus consuming physical (e.g. CPU cycles) and logical resources (e.g. semaphores, OS task structures). Selected locations for fault injection were the `on_run()` method (which is invoked by the platform after the application deployment, in order to create more sophisticated scenarios), the servant CORBA object implementation (the remote method which is invoked by the clients) or `set_state()` method (which is called after a client request is serviced, in order to synchronize primary server's state with the backup replicas); the latter is a critical fault location, in which a fault may compromise the servers' state consistency (a backup server should assume the same state of the primary server just before a failure): inconsistencies can be avoided only using a reliable protocol such as the two-phase commit protocol.

5.5.2 Results

The analysis of the events' traces and of failure detector's coverage supports our initial hyphotesis about most common failure modes, i.e. a remarkable I/O throughput variation from the estimated value is a common failure symptom. In the case of the considered workload, the occurrence of a failure lead to throughput losses both for crashes (figure 5.7) and hangs (figure 5.8). Because of representativeness of the selected failures, the correct detection of these symptoms is an useful tool to design more robust systems using fault treatment techniques against these failure modes.

In figures 5.9a and 5.9b, it is shown the coverage of the failure detector and the FT Manager provided by the middleware. The operating system was



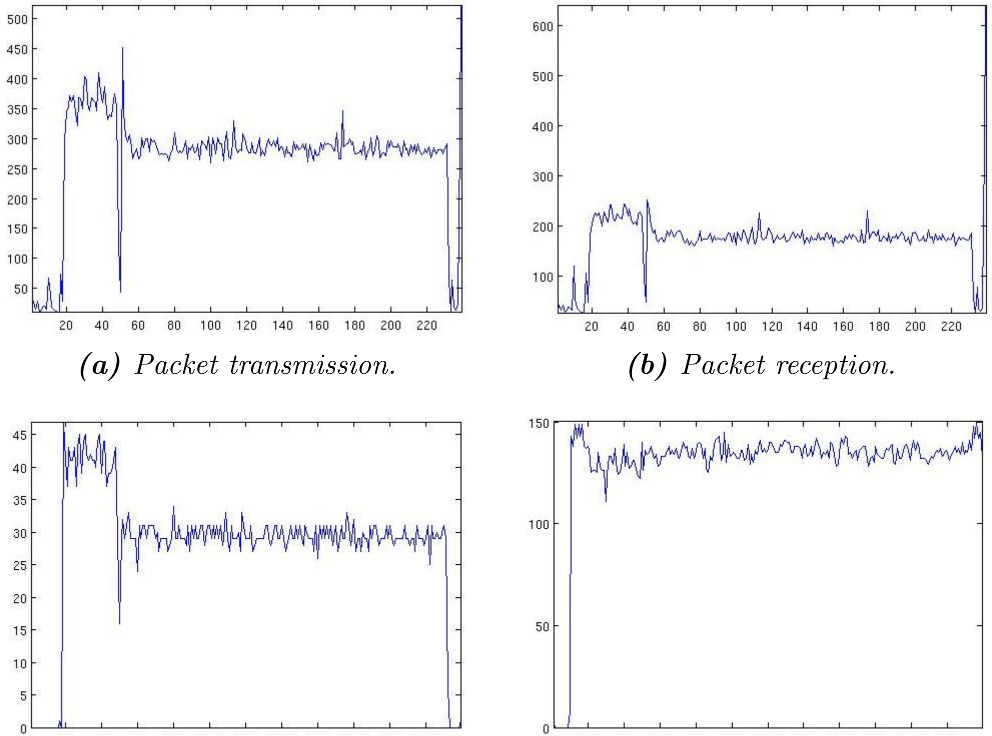
(a) *Packet transmission.*

(b) *Packet reception.*

(c) *Disk reads.*

(d) *Disk writes.*

Figure 5.7: *Observed I/O throughput (with a sampling period $T = 1$ s) during a crash failure experiment. The fault is injected after 40 seconds, in which the server process crashes.*



(a) *Packet transmission.*

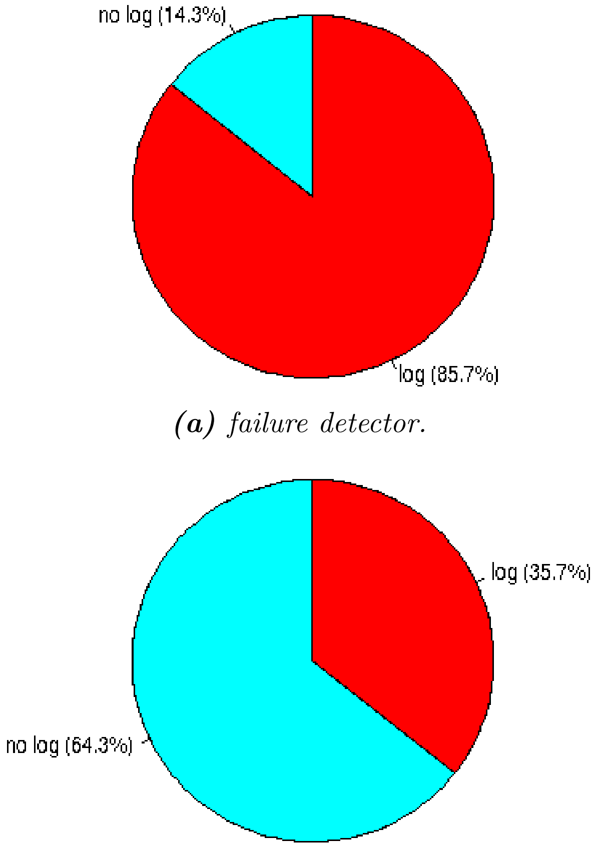
(b) *Packet reception.*

(c) *Disk reads.*

(d) *Disk writes.*

Figure 5.8: *Observed I/O throughput (with a sampling period $T = 1$ s) during an hang failure experiment. The fault is injected after 40 seconds, in which a thread of the server process is indefinitely stalled.*

not subject to error propagation from the application level, therefore no logs were produced about the correctness of its state; from the OS point of view, the processes did execute correctly, and possible failures have to be seek in the semantics of the application (i.e. the coherence of the behavior to the system's specification).



(a) failure detector.

(b) FT Manager.

Figure 5.9: *Log coverage for injections at the application level. All tests logged by the FT Manager were also logged by the failure detector.*

The failure detector provided a greater coverage than the FT Manager. All detected failures were logged with a low latency (lower than 1 second). The cases not logged by the FT Manager and detected by our algorithm were for the most part hangs at the application logical level; because no assumption is made about the behavior of the application, the FT Manager can not distinguish between a misbehaving application and a correct one. When a failure does not lead to the interruption of a process (e.g. a crash) or a denial of the service, or the application does not offer any explicit hint about its erroneous state (e.g. a log file produced by the application), the FT Manager has no way to detect the occurrence of a failure. Nevertheless, the assumption of a throughput variation during a failure is not met in all cases: in two tests (HangOverloading1 and ServerFailure6), the failure did not cause a remarkable difference from the expected throughput, although a performance degradation was observed.

It should be noted that the measured coverage does not necessarily coincide with the coverage that the failure detector will provide for the system in hand. For this purpose, a risk analysis should be carried out in order to estimate the relative occurrence probabilities of the considered failures; then the obtained values can be used to weigh the detector's coverage of each failure mode.

The mean latency for the failure detector is reported in table 5.4. The FT Manager mean latency, which is compared to the one of the failure detector, is evaluated from those cases in which the latencies were low. There were also cases in which a failure was detected by the FT Manager with a very high latency (e.g. hangs in the `on_run()` method); the FT Manager was able to detect the failure because a maximum time for the execution of the behavioral methods was defined in the application logic during the deployment (in our tests, the timeout was set to 20 seconds).

Table 5.4: *Mean detection latencies for failures induced at the user application level. The failure detector samples the I/O throughput with a sampling period of 1 second, in which the detection latency ranges. For the mean latency of the FT Manager, high latency cases (higher than 1 second) were not considered.*

	Failure Detector	FT Manager
Mean latency	0.654069 s	0.205902 s

The latencies for the failure detector result higher than the ones for the FT Manager. The failure detector latency is affected by the choice of the throughput sampling period T (in this case 1 second); if Δ is the mean time for the periodical execution of the algorithm (including the computation of time averages and the threeshold comparisons), then the mean detection latency after the occurrence of a failure is likely to be included in the range $[\Delta, \Delta + T]$. The FT Manager was not affected by the delay caused by periodical sampling, and the detected failures were timely notified (e.g. the explicit exit of a process is immediately notified by the Platform Daemon on the same host).

Therefore, explicit notifications from the application (e.g. application logs) are the most effective way to detect the occurrence of a failure. Nevertheless, due to the absence or the incompleteness of application's detection mechanisms, the proposed failure detector is an effective and cheap way to increase the detection abilities of an already existing application, still providing short notification times.

5.6 Analysis of false positives

In the previous sections, the probability of correct failure detection by the algorithm was evaluated; it is shown that, for common failure modes, the detector produces a notification when a failure occurs, with a reasonable coverage level. However, there is also a significant probability that a failure notification is produced when a failure did not occur, namely a false positive. The occurrence of a false positive is due to a difference between the current sampled throughput and the throughput mean value estimated from previous samples; in turn, the difference can be caused by:

- A random fluctuation of the sampled value around the statistical mean (this event has a non-zero probability $\mathcal{P}(\mathit{ThDiff} \mid \mathit{not\ Failure})$, see § 4.3);

- A sudden and significant variation of the probability distribution at the current time, i.e. the I/O load submitted to the system varies within a period in the order of the sampling period T.

The occurrences of the first cause can be suitably reduced by a correct tuning the algorithm parameters, i.e. by setting a sampling period T that can moderate the oscillations of the observed samples due to secondary factors, and a k threeshold value enough conservative. Nevertheless, the tuning of the algorithm is not guaranteed to be optimal, and the second cause may not be prevented by an adequate tuning, because I/O load variations may not be known during the pre-operational phase, and the throughput is not guaranteed to be approximately stationary (i.e. the mean can be estimated by previous samples) within a time period of $N \cdot T$ seconds.

In order to discriminate between false and correct failure notifications, further processing of detector logs should follow. In this section, we closely analyze the occurrences of false positives, in order to evaluate the extent of wrong notifications, and to propose useful hints for discrimination.

The workload considered for the analysis of false positives was similar to the one used for the assessment of detection coverage and latency; the application was slightly modified, removing the bound on the maximum number of requests. No faultload was defined for this test, because its purpose is to observe the false positives produced by the algorithm during non-faulty execution, in which it should not produce notifications. The test executed for 64 hours, and 3460583 requests were served.

The failure detector produced 3423 log entries, with an average rate of 53.5 entries per hour. The detector would not have produced so much false positives, if the value of k for the network events was higher. The value $k = 7$ reduces the amount of failure positives to 2271, with an average rate of 35.5 entries per hour; this value of k still allows the detection of failures induced by fault injection experiments of previous sections. The number of false positive for each event type is shown in table 5.5, which we will consider in the following of this section.

There were no false positives due to disk reads events; because of read-ahead and buffering mechanisms of the operating system, the number of reads from files was greatly reduced, and so the amount of counted events was too low to trigger the failure detector. The network reception events produced a much higher number of false positives; it can be seen from figure 4.12d that events for this category present a greater variability, which leaded to unexpected throughput differences; this is probably caused by a not good choice of T, and an higher value (e.g. $T = 2$ s instead of $T = 1$ s), keeping

Table 5.5: *The distribution of false positives produced by the detection algorithm during a non-faulty execution period of 64 hours. The algorithm's parameters were $T = 1$ s, $N = 30$, $k = 7$ for network events, and $k = 5$ for disk events.*

	Number of events	Average rate (events/hour)
Network reception	1366	21.4
Network transmission	447	7
Disk reads	0	0
Disk output	458	7.1
Total	2271	35.5

fixed the product $N \cdot T$ (e.g. $N = 15$ instead of $N = 30$), would dampen spurious variations.

We also observed a difference between false positives and correct detections, in term of quantitative variations of the sampled throughput from the estimated mean value. In table 5.6, there are the measured throughput differences (in term of multiples of the estimated standard deviation) which caused a detector notification. The false positives present a lower mean value, although there were several cases in which the throughput difference of a correct failure notification was lower than differences which caused false positives, therefore the correct notifications can not be discriminated by simply comparing the value of the throughput difference to this mean value. It should be also noted that the greater standard deviation of throughput differences for false positives is caused by a relatively small set of cases in which the difference was much greater than the mean value (e.g. $k > 20$ and $k > 30$).

Finally, we examinate the time periods between consecutive false positives, namely *interarrival times.* In figures 5.10a, 5.10b, and 5.10c, it is shown, for each period length, the number of false positives following the previous ones after that period length. It is clearly visible that the most of false positive repeats periodically after a fixed amount of time; for disk writes and network transmission, many notifications repeat every 10 minutes,

Table 5.6: *Mean and standard deviation of throughput differences (i.e. differences between the current throughput sample and the estimated mean value) notified by the algorithm. The values are expressed as multiples k of the current estimated standard deviation.*

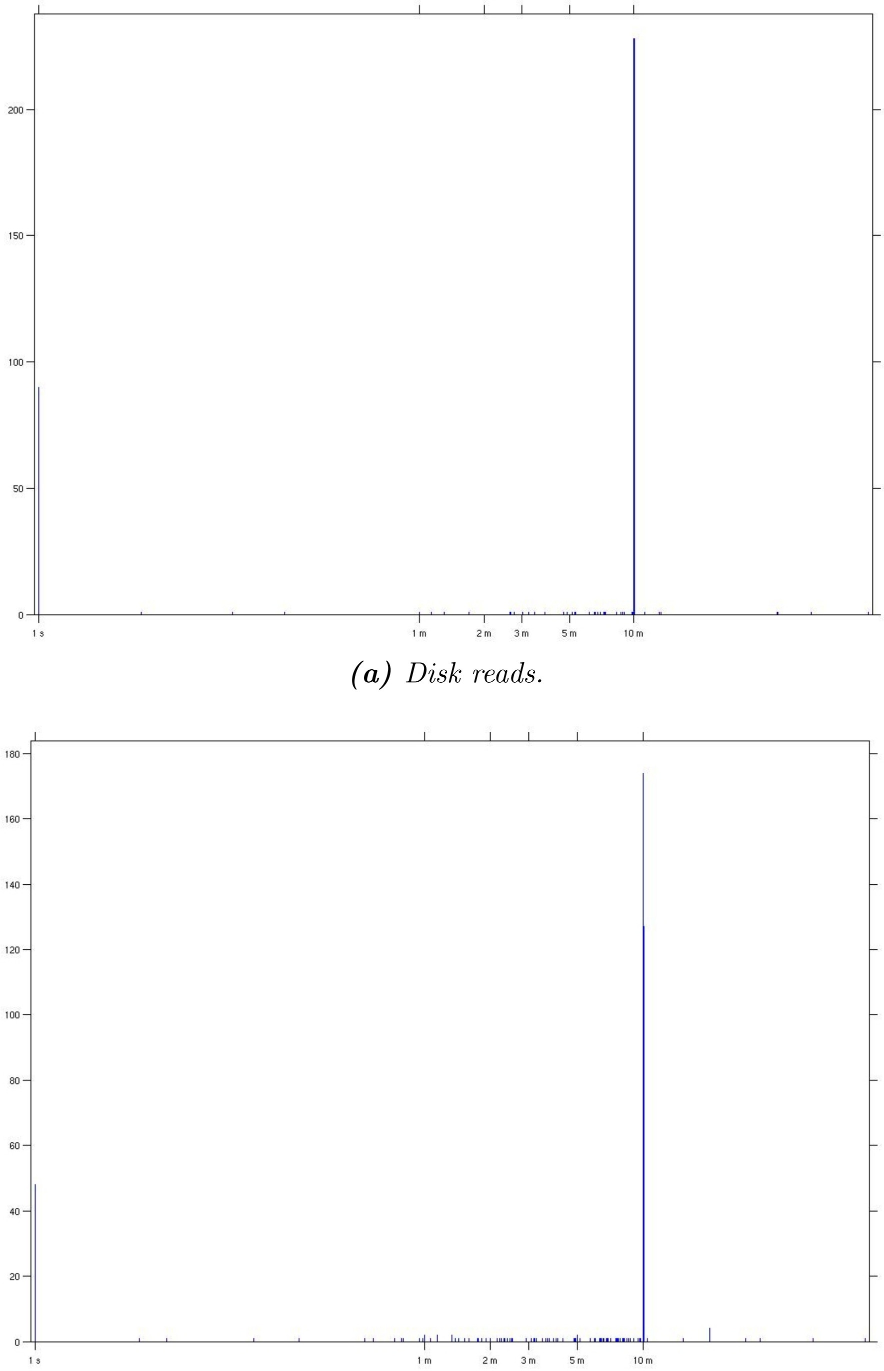
	Application faults	False positives	False positives, excluding $k > 20$ (5/100 cases)	False positives, excluding $k > 30$ (1/100 cases)
Mean	14.909729	10.935000	10.189820	9.555996
Std. dev.	9.438491	9.966790	4.081204	2.930750

and many other immediately follow the previous after 1 second; for network reception, the notifications' repetitions exhibit more complex patterns, although many notifications repeat regularly after 1, 2 and 3 minutes; several interarrival waits between periodic notifications are probably intermixed with random occurrences of false positives, breaking down interarrival times.

The large amount of periodic notifications is due to the occurrence of regular variations of the I/O load, caused by the operating system (e.g. flushing of dirty pages, scheduled maintenance operations) or the application itself (although clients' requests frequency is variable, the same executable code repeats itself indefinitely); we refer to these causes as *systematic factors*, which can be potentially exploited to recognize false positives. In future work, automatic identification of notifications due to systematic factors can be designed, based on the analysis of detector's behavior during a long time period, as we manually made in this section.

5.7 Conclusions

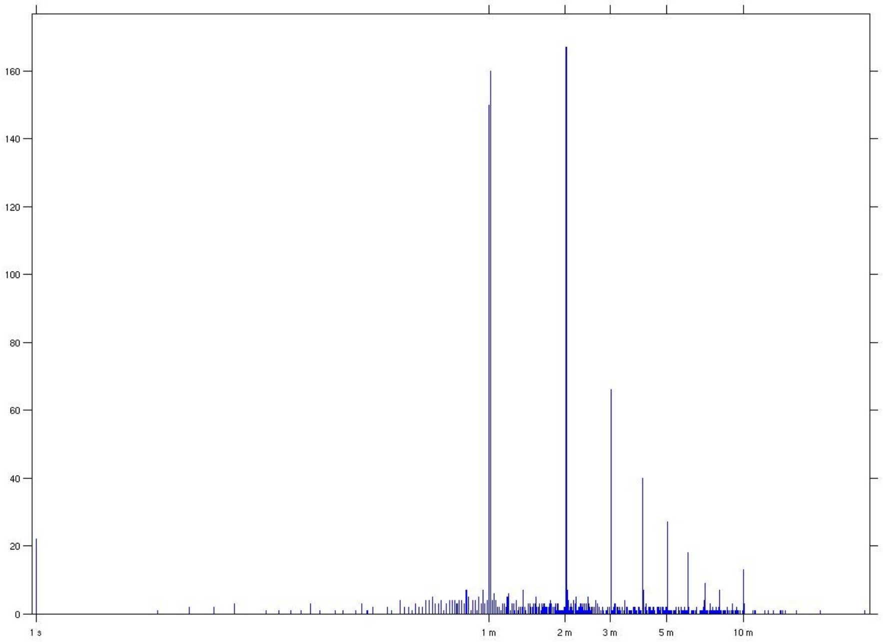
In this chapter, we examined the performances of the proposed algorithm for failure detection, using a workload and a faultload representative of critical applications. For failures induced by the operating system, the detector identified only a relatively small part of failures; nevertheless, the detector improved the coverage of system's logs through the identification of failures



(a) Disk reads.

(b) Network transmission.

Figure 5.10: *Histograms of interarrival times between consecutive notifications.*



(c) Network reception.

Figure 5.10: *Histograms of interarrival times between consecutive notifications (continued).*

not covered by them. Moreover, if the analysis is restricted to system-level failures, the improvement of the failure detector is greater: in general, the failure detector is able to identify only those failures affecting overall performances, and this is not the case of drivers' failures tolerated (and therefore not propagated to the workload) by the operating system. It was also noted that, when a system-level failure due to the operating system occurs, in several cases there is not opportunity for logging or recovery interventions, because of complete interruption of kernel execution (e.g. a kernel panic).

In the case of failures induced by faults at the application level, the failure detector showed an high coverage level, and its logs can be joined with the standard application mechanisms for detection (in this context, the FT Service), with a significant coverage improvement and without any change in the application. The high measured coverage supports our initial hypothesis that, for common failure modes, a remarkable throughput variation is likely to occur.

For both faultloads at the operating system and at the application levels, the failure detector provided a low latency, i.e. the logging delay from the time of injection of a fault was less than or about 1 second. In general, it is expected that the mean latency value amounts to about $T/2$, where T is the throughput sampling period of the algorithm. Therefore, the detector's latency can be considered timely enough to allow potential recovery and fault treatment techniques.

The main obstacle to the utilization of the failure detector on a production environment is its proneness to the production of false positives. The execution on a long time period of the workload without faults, monitored by the failure detector, showed an high false positive rate. The analysis of detector's logs discovered that a great part of false positive can be prevented by a better tuning of algorithm's parameters, and that several notifications were caused by systematic factors that periodically produced variations of the I/O load submitted to the system. Therefore, the suitability of the failure detector for utilization in real-world scenarios depends on other mechanisms that can decide, on the basis of secondary information and further processing of detector and system logs, if a failure actually occurred, and what intervention can be made.

Conclusions and future work

In this work, the problem of the failure detection for complex, COTS-based systems has been discussed. We adopted an empirical approach to study what are the effects of failures, due to software faults, on the execution of the system. We observed that the occurrence of a failure has a remarkable influence on the throughput of I/O operations, for example packets transmitted and received through a network device, or blocks read from and written to a disk. On the basis of the observations on collected data, we defined an algorithm for on-line detection of failures, which exploits throughput measurements to identify an anomaly in device drivers' I/O operations, by comparing a throughput sample to the estimated statistical mean and standard deviation.

In order to evaluate the efficacy of the proposed algorithm and of the existing logging mechanisms, we introduced a dependability benchmark framework for quantitative analysis through experiments. The benchmark assumes as error model the exchange of a wrong value from a device driver to other kernel subsystem, which was also used in several work in the past for robustness testing of operating systems, and to emulate the occurrence of a transient fault in the device drivers (which are the major part of the kernel source code, and have an higher number of faults than other subsystems). The dependability benchmark has been implemented and preliminary experiments were made with a synthetic workload, in order to evaluate its feasibility and reproducibility; moreover, we gained insights into failure modes of the Linux kernel: in particular, it was observed that the logging latencies for different tests can be classified in *low* and *high latencies*, i.e. there exist several cases in which the latency is much more than 1 second (e.g. due to timeouts), and others in which the latency is very low (in the order of hundredths of milliseconds); moreover, we observed that the kernel was able to log only those errors which were tolerated, but no logs were produced when the injection caused a system-level failure.

Then, we used the dependability benchmark for the evaluation of the algorithm under a workload representative of critical applications. In addition to the faultload at the operating system level, we also included software faults at the application level, in order to evaluate the algorithm from more than one point of view. We observed that the algorithm increased the overall log coverage for both types of faults: in the case of operating system's faults, the improvement was less significant because most of faults were tolerated or leaded to the complete interruption of the execution; instead, the algorithm detected the most of faults injected at the application level. Moreover, in all cases the failure detector logged with a low latency. Therefore, the algorithm is able to leverage the initial hypothesis on the throughput variations under failures, to improve logging mechanisms of both the operating system and the middleware platform, guaranteeing bounded latency (in the worst case, the latency do not exceed the sampling period).

The next step, in order to improve system's dependability, will be to merge logs from the failure detector and from system's facilities, and to define recovery mechanisms (at the application or at the operating system level) based on the logs collected. An innovative approach to do so is represented by on-line software diagnosis, which adds a further step between failure detection and the treatment of the fault: the *root cause* of the failure is identified (*isolation*), i.e. the faulty software component and the occurred fault are determined, such that a treatment intervention, focused on the actual fault, can be applied. The detection of a failure on a single node can also be useful in distributed systems, in order to prevent the spread of failures among multiple nodes of the system.

Future work also include further study to improve the failure detector itself. Additional processing on detector's logs is needed in order to rule out false positives: as we observed, there is not a single evidence that allows to distinguish false positives from correct detections, but there exist several features that statistically mark false positives, such as the contiguity of the throughput sample and the threeshold, and the periodicity of the detections' occurrences; those observation can be used to reduce the number of false positives (e.g. by using machine learning algorithms) and to improve detector's effectiveness. Moreover, an automatic procedure for the tuning of the algorithm should be defined, in order to reduce the incidence of human errors, and to make the failure detector self-adapting to variable working conditions (e.g. by forcing the automatic re-tuning of the algorithm when too many false positives occur).

APPENDIX A

Scripts for kernel monitoring and fault injection

A.1 Utilities for dependability benchmarking

In this work, a dependability benchmark is proposed for the assessment of logging mechanisms of both the operating system and the proposed failure detector. The implementation of the dependability benchmark for the case study includes:

- an utility for injection of faults at the operating system level;
- two utilities to identify fault locations for the considered workload;
- an utility for the measurement of the operating system's latency.

As described in section 3.3.1, we assumed as the error model for the operating system the return of a wrong value from a device driver to the kernel, which can be implemented by altering the value of a parameter at the interface between the kernel and device drivers. The injection is made using dynamic probes: a probe is triggered when the target function is invoked, and a parameter (or the return value) is modified by the probe handler (listing A.1). The actual injection is implemented in the C function `inject()` defined in `lib.stp` tapset (see listing A.4):

- the kernel stack is recursively inspected, in order to find the stack frame (i.e. the stack data allocated to a function) referring to the invocation of the target function;
- if the target is a kernel function:
  - a check is made to ensure that the function was called from the device driver's module;
  - if the check is successfull, a function parameter is modified;
- instead, if the target is a driver's function:
  - a check is made to ensure that the function was not called from the driver itself;
  - if the check is successful, the return value of the function is modified;
- if an error was injected, return true, otherwise return false.

When the probe handler ends, the target function's execution is resumed. In all cases, a check is made to verify if the target function is called from a point allowed for injection (i.e. a function exported by the module has to be called by the rest of the kernel, and a function exported by the kernel has to be called by the module). The return address has to belong to the range of allowed addresses, so it is read from the current stack frame (see figure A.1) and compared to the addresses passed to the `inject()` function. In order to identify the current stack frame, a recursive inspection of the stack is made from the top: if the return address of the stack frame i belong to the target function, then the current stack frame at the time of the invocation is the frame $i-1$; if not, the *frame pointer* (i.e. the pointer to the initial address of the stack frame) of the previous stack frame ($i-1$) is retrieved (it is pointed by the frame pointer of the current stack frame), and the check is repeated on the previous stack frame.

In the x86 architecture, the current frame pointer is stored in the `EBP` register. The described stack layout is also used for the kernel stack if it is

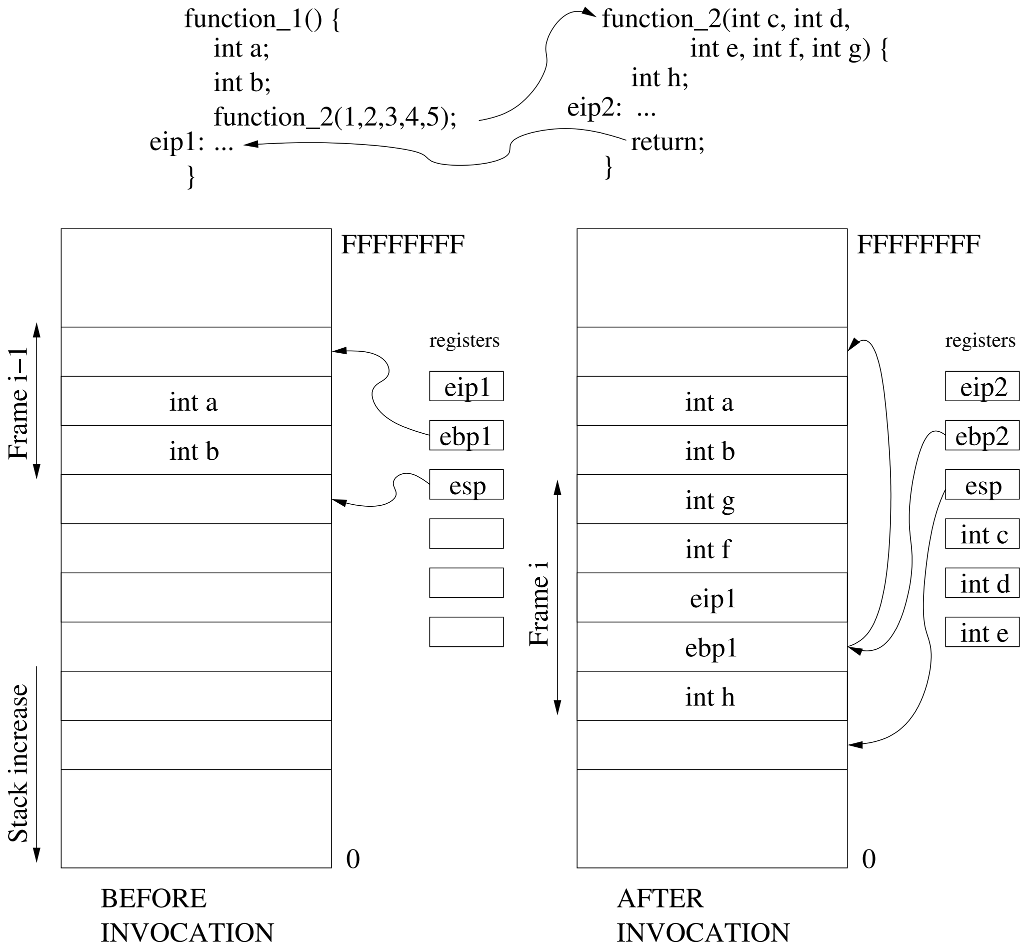


Figure A.1: *The calling convention used in the kernel.*

compiled with the `CONFIG_FRAME_POINTER` option; if not, stack inspection is still possible, although it is slightly more complicated and inefficient. The check of the stack is made each time the probe is activated, even if an error was already injected and does not have to be injected anymore, because we do not want to introduce significant performance differences compared with the period in which an error is injected.

All parameters considered for the injection does fit in a single data word (in x86, the size of a data word is 32 bits): if the parameter is an interger value (e.g. int, long), a char, or a pointer to other data (e.g. a struct or a function), it is directly stored in a register or in a word on the stack. In order to alter invocation's parameters, the probe handler has to access to the current stack frame and to the registers at the time of the invocation. The parameters are passed to a function following this convention:

- the first parameter is passed through the `EAX` register;
- the second parameter is passed through the `EDX` register;
- the third parameter is passed through the `ECX` register;
- the parameters following the third one, if any, are pushed on the stack in *reverse order*, i.e. the last parameter is the first one pushed on the stack, and the fourth parameter is the last one.

Before the execution of the probe handler, the CPU context containing the registers' value at the time of the invocation is saved by the KProbes framework, and it available to the handler through the `CONTEXT` macro. Those value are modified, and the saved context is written to CPU register after the execution of the probe handler. In the case of alteration of a return value, the mechanism for the injection is similar: the calling convention prescribes that a value has to be returned through the `EAX` register, which is modified by the handler.

The `inject()` function is also used for tracing of the functions called from a module, and functions of a module called by the kernel, during the execution of the workload, which represent suitable locations for injection of errors. When the check is successful, `inject()` does not modifies function's parameters, but it still returns a boolean true value, as if it injected an error; the behavior is modified by an dedicated input parameter.

Another script is used for measurement of the latency of logging mechanisms (see listing A.5): the syslogd process is monitored to trace its invocations of the `sys_write()` system call, which is used to store log entries to system files such as `/var/log/messages`; the timestamp of the return from the system call is printed, which can be compared to the timestamp of the injection of the error to evaluate the latency.

The last script (listing A.6) is used to trace scsi and network devices activities. The events are stored in binary format trace, to achieve better performances and lower I/O overhead. The trace is analyzed after workload execution, in order to gain insights into the performances of the system with and without failures.

In the following, we show the source code of the described scripts for dependability benchmarking.

Listing A.1: *SystemTap script for injection of bit-flip errors on the interface between the kernel and device drivers (injector.stp).*

```
/*
   Script name: injector.stp
   Version: 1.0.0
   Description: Utility for injection of bit-flip
      of parameters at the device drivers’ interface

   Usage:
   $ stap injector.stp -g -I DIR_WITH_LIB_STP \
         <func_name> <param_num> <bit_mask> \
         <module_init_addr> <module_end_addr>

   1) the name of the targeted kernel function
   2) the signature’s position of the injected parameter
      (0 to inject in the return value, 1 in the first
      parameter, and so on)
   3) the bit mask to use for bit flip injection
      (radix 10 format)
   4) initial address of the target module text segment
      (radix 16 format)
   5) final address of the target module text segment
```

```
      (radix 16 format)

*/

/*  The total number of consecutive faults to inject  */
global total_injections = 1


/*  The injection is started after a fixed amount
   of seconds  */
global wait_time = 1


global module_start_addr
global module_end_addr
global reject_start_addr
global reject_end_addr
global inject_param
global bitmask

global start_time

global inj_func_addr_start
global inj_func_addr_end

probe begin {
   %( $2>0 %?
      module_start_addr = $4
      module_end_addr = $5
      reject_start_addr = 0
      reject_end_addr = 0
   %:
      module_start_addr = 0
      module_end_addr = 0xFFFFFFFF
      reject_start_addr = $4
      reject_end_addr = $5
   %)

   inject_param = strtol(@2,10)
   bitmask = strtol(@3,10)

   start_time = gettimeofday_s()

   /* Initial and final address of the target
      function's code */
   inj_func_addr_start = get_func_addr_start(@1);
   inj_func_addr_end = get_func_addr_end(@1);
}

probe
%( $2>0 %?
   // Inject a fault in the function call parameters
   module("*").function(@1)?,
   kernel.function(@1)?
%:
   // Inject a fault in the function return value
   module("*").function(@1).return?,
   kernel.function(@1).return?
%)
{
   start_injection = 0;
   if(gettimeofday_s()>=start_time+wait_time &&
      total_injections>0)
   {
      start_injection = 1;
   }

   injected = 0;

   injected = inject(module_start_addr, module_end_addr,
      reject_start_addr, reject_end_addr,
      inject_param, bitmask,
      start_injection,
      inj_func_addr_start, inj_func_addr_end)

   if(start_injection && injected) {
      total_injections--
      printf("%d\n", gettimeofday_ns());

      // Uncomment this to log through syslogd
      // printk_warning("INJECTOR", gettimeofday_ns());
   }
```

```
}
```

Listing A.2: *SystemTap script for monitoring of functions imported by a module (calls_from_module.stp).*

```
/*
   Script name: calls_from_module.stp
   Version: 1.0.0
   Description: Utility for tracing kernel functions
      called by a module

   Usage:
   $ stap calls_from_module.stp -g -I DIR_WITH_LIB_STP
*/

/*
   start_addr and end_addr are the initial and final
   addresses of the TEXT segment of the monitored
   module; they are available in:
   /sys/module/MODULENAME/sections/.text
   /sys/module/MODULENAME/sections/.exit.text
*/

/*
   sample addresses of the ahci module
   for the vanilla kernel compiled by
   ourself
*/
global start_addr = 0xf883e000
global end_addr = 0xf883fa9c

probe callerprobe {

   func_name = probefunc();
   func_start = get_func_addr_start(func_name);
   func_end = get_func_addr_end(func_name);

   if(func_start==0 || func_end==0) {
      next;
   }

   found = check_if_called_by_module(start_addr,end_addr,
                           func_start,func_end);

   if(found) {
      print(probefunc()."\n");
   }

}


probe callerprobe +=
/*
   The functions to monitor have to be singularly
   defined there (kernel functions are too many to be
   all monitored); for example, to monitor all
   functions in the sd_mod and scsi_mod modules
   called by the target module (e.g. ahci):

   module("sd_mod").function("*"),
   module("scsi_mod").function("*"),
   ...

   To monitor all functions exported by the kernel
   and used within the module:

   kernel.function("printk")?,
   kernel.function("memcpy")?,
   kernel.function("kfree")?,
   ...
*/
   {}
```

Listing A.3: *SystemTap script for monitoring of functions exported by a module (calls_to_module.stp).*

```
/*
   Script name: calls_to_module.stp
   Version: 1.0.0
   Description: Utility for tracing functions of
      a module called by the rest of the kernel

   Usage:
   $ stap calls_to_module.stp -g -I DIR_WITH_LIB_STP
*/

/*
   start_addr and end_addr are the initial and final
   addresses of the TEXT segment of the monitored
   module; they are available in:
   /sys/module/MODULENAME/sections/.text
   /sys/module/MODULENAME/sections/.exit.text
*/

/*
   sample addresses of the ahci module
   for the vanilla kernel compiled by
   ourself
*/
global start_addr = 0xf883e000
global end_addr = 0xf883fa9c

probe module("ahci").function("*") {

   func_name = probefunc();
   func_start = get_func_addr_start(func_name);
        func_end = get_func_addr_end(func_name);

   if(func_start==0 || func_end==0) {
      next;
   }

   reject = check_if_called_by_kernel(start_addr, end_addr,
                           func_start, func_end);

   if(reject) {
      print(probefunc()."\n");
   }
}
```

Listing A.4: *A tapset with auxiliary functions (lib.stp).*

```
/*
```

```
   Script name: lib.stp
   Version: 1.0.0
   Description: Tapset with auxiliary functions
      for kernel tracing and fault injection

   Usage: automatically included by other scripts
*/

%{
#include "sym.h"
%}

/* Functions to obtain the initial and final address
   of a function defined in the kernel or in a module */

function __get_func_addr_start:long (func_name:string)
%{
   /* The function's code start address is obtained
      from the SystemTap symbol table build during
      symbols resolution for probe registration
      (see runtime/sym.c SystemTap file) */

   int i;
   for(i=0; i<_stp_num_modules; i++) {
      struct _stp_symbol *s = _stp_modules[i]->symbols;
      unsigned num = _stp_modules[i]->num_symbols;

      while (num--) {
         if (strcmp(THIS->func_name, s->symbol) == 0) {
            THIS->__retvalue = s->addr;
            return;
         }

         s++;
      }
   }

   THIS->__retvalue = 0;
%}

function __get_func_addr_end:long (func_name:string)
```

```
%{
   /* The function's code end address is estimated
      by using the start address of the following
      function in the SystemTap symbol table
      (see runtime/sym.c SystemTap file) */

   int i;
   for(i=0; i<_stp_num_modules; i++) {
      struct _stp_symbol *s = _stp_modules[i]->symbols;
      unsigned num = _stp_modules[i]->num_symbols;

      while (num--) {
         if (strcmp(THIS->func_name, s->symbol) == 0) {
            THIS->__retvalue = (s+1)->addr;
            return;
         }

         s++;
      }
   }

   THIS->__retvalue = 0;
%}

/*
   Function for error injection in the kernel; TRUE is
   returned if the current function is called by a
   function in the range

   [module_start_addr, module_end_addr]

   and NOT in the range

   [reject_start_addr, reject_end_addr]

   If start_injection is TRUE, the frame stack is
   inspected but a fault is not injected, even if all
   other conditions hold. Bitmask is an integer value
   representing the bit-mask used for bit-flip (it is
   XOR-ed with the actual value). Inject_param is an
   integer representing the position of the parameter
   in the function's signature (0 for the return value).
   The current function's code is in the addresses

   [func_addr_start, func_addr_end]

*/

function inject:long (module_start_addr:long, module_end_addr
  ↪ :long, reject_start_addr:long, reject_end_addr:long,
  ↪ inject_param:long, bitmask:long, start_injection:long,
  ↪ func_addr_start:long, func_addr_end:long)
%{

/* see print_context_stack() in arch/i386/kernel/traps.c
   for details about stack inspection in the kernel */

unsigned long ebp = CONTEXT->regs->ebp;
unsigned long *stack = &(CONTEXT->regs->esp);

unsigned long addr = CONTEXT->regs->eip;

/* the stack base address is the THREAD_SIZE-aligned
   address (THREAD_SIZE is 4k or 8k) following the
   current stack pointer */
unsigned long base_stack_address =
   ((unsigned long)stack & (~(THREAD_SIZE - 1)));

/* true if the address belongs to the targeted module
   text segment */
bool found = false;

/* true if the address is in the rejected range */
bool reject = false;

/* maximum stack depth to inspect */
unsigned int max_depth = 15;
unsigned int depth = 0;

unsigned long *stack_top = 0;

unsigned long *param_addr;
```

```

bool ok;

if(THIS->inject_param==0) {

   /* the return address of the current frame context
      (works only for return probes at the time of
      writing - systemtap snapshot 20070623) */
   unsigned long return_addr =
         (unsigned long)CONTEXT->pi->ret_addr;

   /* we are in the return probe handler */

   if(return_addr>=THIS->module_start_addr &&
      return_addr<THIS->module_end_addr)
   {
      found = true;
   }

   if(return_addr>=THIS->reject_start_addr &&
      return_addr<THIS->reject_start_addr)
   {
      reject = true;
   }

} else {

#ifdef CONFIG_FRAME_POINTER

   while (  ebp>=base_stack_address
         && ebp<=base_stack_address+THREAD_SIZE-3
         && depth<max_depth)
   {

      unsigned long new_ebp;

      if(  addr>=THIS->func_addr_start
        && addr<=THIS->func_addr_end)
      {
         /*
            Check if the return address of the
            current stack frame belong to the
            module's executable addresses
         */

         addr = *(unsigned long *)(ebp + 4);

         if(  addr>=THIS->module_start_addr
           && addr<THIS->module_end_addr)
         {
            stack_top = (unsigned long *)(ebp + 8);
            found = true;
         }

         if(  addr>=THIS->reject_start_addr
           && addr<THIS->reject_end_addr)
         {
            reject = true;
         }

         break;
      }

      /* The return address of the current stack frame */
      addr = *(unsigned long *)(ebp + 4);

      /*
         break if the next ebp address is lower than
         the current one (because previous stack frames
         are located at increasing addresses)
      */

      new_ebp = *(unsigned long *)ebp;
      if (new_ebp <= ebp)
         break;

      ebp = new_ebp;

      depth++;

   }

```

```
#else
#error Please, compile the kernel with the CONFIG_FRAME_POINTER option enabled
#endif

}

ok = found && !reject;

THIS->__retvalue = ok;

if(THIS->start_injection==0) {
   return;
}

if(ok) {

/*
   if inject_param>0 and the targeted module has called
   the targeted function, inject a fault in the parameter;
   if inject_param==0 and a function of the targeted
   module is returning, inject a fault in the return value
*/


      switch(THIS->inject_param) {
         case 0:
      /*
         we assume as calling convention that the
         return value (if any) is provided through
         %eax for 32 bit types (e.g. int)
      */
            CONTEXT->regs->eax =
               CONTEXT->regs->eax ^ THIS->bitmask;
            break;

      /*
         If the targeted parameter is the first, the
         second or the third, it is stored in a register
         before the call (in order, %eax, %edx or %ecx);
         we assume the kernel was compiled with the
         -mregparam=3 option, that sets subroutines'
         calling convention
      */
         case 1:
            CONTEXT->regs->eax =
               CONTEXT->regs->eax ^ THIS->bitmask;
            break;
         case 2:
            CONTEXT->regs->edx =
               CONTEXT->regs->edx ^ THIS->bitmask;
            break;
         case 3:
            CONTEXT->regs->ecx =
               CONTEXT->regs->ecx ^ THIS->bitmask;
            break;

      /*
         The caller has pushed the targeted parameter
         on the stack (the last parameter is the first
         to be pushed, and the fourth parameter is the
         last)
      */
         default:
      /*
         we evaluate the address of the top element of
         the stack just before the call of the targeted
         function; we use the address of the return
         address found in the previous code when scanning
         the call frames on the stack (stack_top):
         function's parameters were push just before
         the return address we have found
      */
         param_addr =
               stack_top + 4*(THIS->inject_param-3)-4;

         *param_addr = *param_addr ^ THIS->bitmask;

         break;
      }

}
```

```

%}


/* static variables for caching functions'
   address ranges */
global __func_addr_start
global __func_addr_end
global __func_inline_excluded

/* get the initial address of the function's code */
function get_func_addr_start:long (func_name:string) {
   if(__func_inline_excluded[func_name]==1) {
      return 0;
   }

   func_start = __func_addr_start[func_name];

   if(func_start==0) {
      func_start = __get_func_addr_start(func_name);

      if(func_start==0) {
         __func_inline_excluded[func_name] = 1;
         return 0;
      } else {
         __func_addr_start[func_name] = func_start;
      }
   }

   return func_start
}

/* get the final address of the function's code */
function get_func_addr_end:long (func_name:string) {
   if(__func_inline_excluded[func_name]==1) {
      return 0;
   }

   func_end = __func_addr_end[func_name];

   if(func_end==0) {
      func_end = __get_func_addr_end(func_name);

      if(func_end==0) {
         __func_inline_excluded[func_name] = 1;
         return 0;
      } else {
         __func_addr_end[func_name] = func_end;
      }
   }

   return func_end
}

/* check if the current function is a kernel function
   called by the specified module */
function check_if_called_by_module:long (start_addr:long,
  ↪ end_addr:long, func_start:long, func_end:long)
{
   return inject(start_addr, end_addr,
               0, 0,    // do not reject
               1, 0, 0,   // no injection
               func_start, func_end
            );
}

/* check if the current function is a function of the
   specified module called by the kernel */
function check_if_called_by_kernel:long (start_addr:long,
  ↪ end_addr:long, func_start:long, func_end:long)
{
   return inject(0, 4294967295,  // any caller function
               start_addr, end_addr,
               1, 0, 0,    // no injection
               func_start, func_end
            );
}

/* print in syslogd log files
   (e.g. /var/log/messages, /dev/console) */
function printk_warning(prefix:string, msg:string) %{
   printk(KERN_WARNING "%s %s\n", THIS->prefix, THIS->msg);
```

```
%}
```

Listing A.5: *SystemTap script for measurement of logging latency by the Syslogd daemon (syslog-write.stp).*

```
/*
   Script name: syslog-write.stp
   Version: 1.0.0
   Description: Utility for measurement of syslogd
      logging latency

   Usage:
   $ stap syslog-write.stp -x ‘pidof syslogd‘
*/

probe syscall.writev.return, syscall.write.return {
   if(pid()==target()){
      printf("%d\n", gettimeofday_ns());
   }
}
```

Listing A.6: *SystemTap script for tracing I/O activities of network and disk devices drivers (trace.stp).*

```
/*
   Script name: trace.stp
   Version: 1.0.0
   Description: Utility for tracing kernel I/O events

   Usage:
   $ stap trace.stp -bMv
*/

probe begin
{
  stoptrace_fork = 1
  stoptrace_exec = 1
}

probe addevent.scsi.iodone {}
probe addevent.netdev.receive {}
probe addevent.netdev.transmit {}
```

A.2 Implementation of the failure detector

The proposed algorithm for failure detection was implemented with the SystemTap language (see § 5.3), to be experimentally evaluated through the dependability benchmark for logging mechanisms which was defined in section 3.3.1.

The failure detector consists of 3 probes to monitor packets transmission and reception, and disk block requests (reads and writes are discriminated by inspection of a flag); a timer probe is defined for the periodic evaluation of the inequality (4.13). When the timer activates the probe, the statistical mean and the standard deviation for the current sample are estimated from the previous samples, and a log entry is produced (it is sent to the standard output of the `stap` process, or to `syslogd` through the `printk()` kernel function), which describes the throughput variation (e.g. what event caused the variation, the estimated values, the current sample, and so on). The test on the current sample is repeated for each one of the 4 monitored I/O events (packet send/receive, block read/write).

Because of the fact we do not use floating point arithmetic in kernel code (we should explicitly access to the floating point registers of the CPU and manage interrupts caused by operations; moreover, the floating point arithmetic requires more CPU cycles), the failure detector does not directly implement (4.13), (4.14), and (4.15). Instead, the following expressions are implemented, by only using integer arithmetic:

$$(X(t_i) - m_X)^2 - k^2\sigma_X^2 \geqslant 0 \tag{A.1}$$

$$m_X = \left\lfloor \frac{\sum_{k=i-N}^{i-1} \overline{x}_k}{N} \right\rfloor \tag{A.2}$$

$$\sigma_X^2 = \left\lfloor \frac{\sum_{k=i-N}^{i-1} (\overline{x}_k - m_X)^2}{N-1} \right\rfloor \tag{A.3}$$

At the time of writing, only integer values for the k parameter are supported. Nevertheless, the (A.1) can be rearranged by multiplying the left member with a constant factor, in order to support non-integer values of k with arbitrary precision. For example, if the expression is multiplied for 100, the second term of the subtraction can be written as $100 \cdot k^2 \sigma_X^2 = (10 \cdot k)^2 \sigma_X^2 = \overline{k}^2 \sigma_X^2$, therefore if the desired value of k is 6.5, we will assign the value 65 to the variable $\overline{k}$.

The sampled throughput values are stored in a circular buffer (figure A.2): the current sample location is pointed by an integer index, and previous N samples are stored in previous locations (modulo N+1). After that the periodic processing of samples is completed, the pointer is increased (modulo N+1), the pointed location is zeroed, and all probes monitoring I/O events will increase the value of the circular buffer at that location.

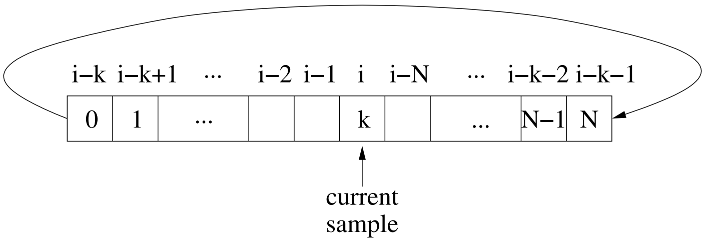


Figure A.2: *The circular buffer data structure for throughput samples.*

In the following, the source code of the described failure detector is reported.

Listing A.7: *SystemTap script for failure detection (detect.stp).*

```
/*
   Script name: detect.stp
   Version: 1.0.0
   Description: Utility for failure detection

   Usage:
   $ stap detect.stp -g -DMAXACTION=100000 \
      <net interfaces list> <disks list> <sampling_time>

   The first and second parameter are semicolon-separated
```

```
lists containing, respectively, the network interfaces
and disk devices to monitor; the throughput of each
monitored device is periodically sampled, and the
number of periods between consecutive samples should
be specified in the lists for each device. The third
parameter is the length of a single period in
milliseconds. Each item in the lists contains:

   * The device identifier (net interface name or
     disk major and minor number);
   * The number of periods between samples for input
     data throughput (read blocks for disk, input
     packets for net);
   * The number of periods between samples for output
     data throughput (written blocks for disk, output
     packets for net);
   * The number N of previous samples, for input data,
     compared to the current sample by the detection
     algorithm;
   * The number N of previous samples, for output
     data;
   * The number k, for input data, which is multiplied
     with the standard deviation to obtain a range
     around the mean of the N previous samples; the
     sampled throughput should exceed this range only
     in the case of failures;
   * The number k, for output data.

Both N and k should be integer numbers.

In the following example, the network interfaces eth0
and eth1 are monitored with a sample period (both input
and output) of 100 ms (1 period of 100 ms), and the
disk devices sda and hda with a sample period (both
input and output) of 500 ms (5 periods of 100 ms).
The N parameter is 10 for each device (both input and
output), and the k parameter is 3 for the network
interfaces and 4 for the disks (both input and output).

$ detect.stp "eth0,1,1,10,10,3,3;eth1,1,1,10,10,3,3" \
             "sda,5,5,10,10,4,4;hda,5,5,10,10,4,4"    \
                100

   To only monitor the eth0 network interface:

   $ detect.stp "eth0,1,1,10,10,3,3" "" 100
*/



/*
   the list of network devices
*/
global net_devices

/*
   the list of disk devices
*/
global disk_devices

/*
   the number of periods between samples for each device
*/
global samples
global past_samples

/*
   bidimensional matrixes containing circular buffers
   of size N+1
*/
global counters

/*
   arrays of logical pointers to each current entry
   in the circular buffers in counters
*/
global index

/*
   "N" contains the number of previous samples for each
   device to use in order to evaluate the mean and
   variance of processed blocks/packets (the N parameter)
*/
```

```
global N

/*
   "k" contains the k parameter for each monitored device
*/
global k

/*
   The minimum number of events needed to evaluate
   statistics for the drivers; if the number of
   events is lower than "threeshold", a warning can
   not be produced
*/
global threeshold = 100


global period = $3
global input_index = 1
global output_index = 2

global frequency

/*
   Temporary arrays for lists parsing
*/
global net_array
global disk_array


global RW_MASK


probe begin {
   parse_lists()

   foreach(dev in net_devices) {
      for(i=0; i<N[dev,input_index]; i++) {
         counters[dev,input_index,i] = 0
         index[dev,input_index] = 0
      }

      for(i=0; i<N[dev,output_index]; i++) {
         counters[dev,output_index,i] = 0
         index[dev,output_index] = 0
      }
   }

   foreach(dev in disk_devices) {
      for(i=0; i<N[dev,input_index]; i++) {
         counters[dev,input_index,i] = 0
         index[dev,input_index] = 0
      }

      for(i=0; i<N[dev,output_index]; i++) {
         counters[dev,output_index,i] = 0
         index[dev,output_index] = 0
      }
   }

   frequency = 1000 / period

   RW_MASK = get_rw_mask();
}



probe timer.ms($3) {

   foreach([dev, inout] in samples) {

      numsamples = samples[dev, inout]

      past_samples[dev, inout] =
         (past_samples[dev, inout] + 1) % numsamples

      if(past_samples[dev,inout]!=0) {
         continue
      }

      idx = index[dev,inout]
      count = counters[dev,inout,idx]
      numperiods = N[dev,inout]
```

```
    bufsize = numperiods + 1

    /*
       the throughput for the current sample
    */
    throughputshort = count * frequency / numsamples

    /*
       the throughput for the previous N samples
    */
    throughputlong = 0;

    for(j=0; j<numperiods; j++) {
       throughputlong +=
          counters[dev, inout, (idx+1+j)%bufsize]
    }

    total = throughputlong + counters[dev, inout, idx]

    throughputlong =
       throughputlong*frequency/(numsamples*numperiods)

    /*
       variance evaluated on the previous N samples
    */
    var = 0

    for(j=0; j<numperiods; j++) {
       count = counters[dev, inout, (idx+1+j)%bufsize]
       sample = (count * frequency / numsamples)
              - throughputlong

       var += sample*sample
    }

    var = var / (numperiods-1)

    k_i = k[dev,inout]

    mean_diff = throughputshort - throughputlong
```

```
      eval = mean_diff*mean_diff - k_i*k_i*var

      if(total >= threeshold) {

         if(eval > 0) {

            source = ""
            warn_msg = ""

            if(net_devices[dev]==1) {
               source = "network_card_"
            } else if(disk_devices[dev]==1) {
               source = "disk_"
            }

            if(inout==input_index) {
               source = source."input"
            } else {
               source = source."output"
            }

            if(mean_diff >= 0) {
               warn_msg = "throughput_peak"
            } else {
               warn_msg = "throughput_loss"
            }

            log_entry = sprintf("%d %s:%s %s\tthshort=%d,
             ↪ thlong=%d, var=%d, k=%d", gettimeofday_ns(),
             ↪ source, dev, warn_msg, throughputshort,
             ↪ throughputlong, var, k_i);

            print(log_entry."\n");

            // Uncomment this to log through syslogd
            //printk_warning("DETECTOR", log_entry);
         }
      }

      /*
         Reset the next buffer entry for the next
         throughput sample
      */
      counters[dev, inout, (idx+1)%bufsize] = 0

      /*
         Increase the current entry index
      */
      index[dev,inout] = (idx+1)%bufsize
   }
}

function printk_warning(prefix:string, msg:string) %{
   printk(KERN_WARNING "%s %s\n", THIS->prefix, THIS->msg);
%}

probe netdev.transmit {
   if(net_devices[dev_name]==1) {
      idx = index[dev_name, output_index]
      counters[dev_name, output_index, idx]++
   }
}

probe netdev.receive {
   if(net_devices[dev_name]==1) {
      idx = index[dev_name, input_index]
      counters[dev_name, input_index, idx]++
   }
}

probe ioscheduler.elv_completed_request {

   if($rq!=0 && $rq->rq_disk!=0) {

      /*
         see struct request in include/linux/blkdev.h
         and include/linux/genhd.h
      */
      disk_name = kernel_string($rq->rq_disk->disk_name)

      if(disk_devices[disk_name]==1) {

         if(req_flags & RW_MASK) {
            // write request
            idx = index[disk_name, output_index]
            counters[disk_name, output_index,idx]++
         } else {
            // read request
            idx = index[disk_name, input_index]
            counters[disk_name, input_index,idx]++
         }

      }

   }

}

/*
   The REQ_RW mask is defined in
   include/linux/blkdev.h
*/

%{
#include <linux/blkdev.h>
%}

function get_rw_mask:long() %{
   THIS->__retvalue = REQ_RW;
%}


function parse_lists() {
   net_list = @1
   disk_list = @2

   delete net_array
   delete disk_array

   if(net_list=="" && disk_list=="") {
      error("There must be at least one device in the net
       ↪ interfaces list or disks list")
   }
```

```

// Parse the network interface list
i = 0

net_item = tokenize(net_list,";")

if(net_list=="") {
   /*
      do nothing; it means that only
      disk devices were specified by
      the user
   */
} else if(net_item=="") {
   error("The first parameter is not a valid list of
     ↪ network interfaces")
} else {

   while(net_item != "") {
      net_array[i] = net_item
      i++

      net_item = tokenize("",";")

   }

   for(j=0; j<i; j++) {
      if_name = tokenize(net_array[j],",")

      sample_periods_in = tokenize("",",")
      sample_periods_out = tokenize("",",")

      numperiods_in = tokenize("",",")
      numperiods_out = tokenize("",",")

      k_in = tokenize("",",")
      k_out = tokenize("",",")

      if(if_name=="" || sample_periods_in=="" ||
         sample_periods_out=="" ||
         numperiods_in=="" || numperiods_out=="" ||
         k_in=="" || k_out=="" ) {
```

```
         error("The first parameter is not a valid list of network interfaces")
      }

      net_devices[if_name] = 1

      samples[if_name, input_index] =
         strtol(sample_periods_in,10)
      samples[if_name, output_index] =
         strtol(sample_periods_out,10)

      past_samples[if_name, input_index] = 0
      past_samples[if_name, output_index] = 0

      N[if_name, input_index] =
         strtol(numperiods_in,10)
      N[if_name, output_index] =
         strtol(numperiods_out,10)

      k[if_name, input_index] = strtol(k_in,10)
      k[if_name, output_index] = strtol(k_out,10)
   }

}

// Parse the disk devices list
i = 0;

disk_item = tokenize(disk_list,";")

if(disk_list=="") {
   /*
      do nothing; it means that only
      net devices were specified by
      the user
   */
} else if(disk_item=="") {
   error("The second parameter is not a valid list of disk devices")
} else {

   while(disk_item != "") {
      disk_array[i] = disk_item
      i++

      disk_item = tokenize("",";")

   }

   for(j=0; j<i; j++) {
      disk_name = tokenize(disk_array[j],",")

      sample_periods_in = tokenize("",",")
      sample_periods_out = tokenize("",",")

      numperiods_in = tokenize("",",")
      numperiods_out = tokenize("",",")

      k_in = tokenize("",",")
      k_out = tokenize("",",")

      if(disk_name=="" || sample_periods_in=="" ||
         sample_periods_out=="" ||
         numperiods_in=="" || numperiods_out=="" ||
         k_in=="" || k_out=="" ) {
         error("The second parameter is not a valid list
           ↪ of disk devices")
      }

      disk_devices[disk_name] = 1

      samples[disk_name, input_index] =
         strtol(sample_periods_in,10)
      samples[disk_name, output_index] =
         strtol(sample_periods_out,10)

      past_samples[disk_name, input_index] = 0
      past_samples[disk_name, output_index] = 0

      N[disk_name, input_index] =
         strtol(numperiods_in,10)
      N[disk_name, output_index] =
         strtol(numperiods_out,10)

      k[disk_name, input_index] = strtol(k_in,10)
      k[disk_name, output_index] = strtol(k_out,10)
   }

}
}
```

References

There are no answers, only cross-references.

Norber Weiner

References from articles

[AAF04] Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 867, Washington, DC, USA, 2004. IEEE Computer Society.

[ACC$^+$91] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. Technical Report 91260, LAAS-CNRS, 1991.

[AJM$^+$06] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for Directed I/O. *Intel Technology Journal*, 2006.

[BCCR04] Andrea Bondavalli, Silvano Chiaradonna, Domenico Cotroneo, and Luigi Romano. Effective fault treatment for improving the dependability of COTS and legacy-based applications. *IEEE Trans. Dependable Secur. Comput.*, 1(4):223–237, 2004.

[BCFV07] Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai, and Michele Vadursi. Foundations of measurement theory applied to the evaluation of dependability attributes. In *DSN*, pages 522–533. IEEE Computer Society, 2007.

[BS95] Michael F. Buckley and Daniel P. Siewiorek. VAX/VMS event monitoring and analysis. In *FTCS*, pages 414–423, 1995.

[BS96] Michael F. Buckley and Daniel P. Siewiorek. A comparative analysis of event tupling schemes. In *FTCS*, pages 294–303, 1996.

[CBC+92] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal Defect Classification-a concept for in-process measurements. *IEEE Trans. Softw. Eng.*, 18(11):943–956, 1992.

[CC96] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *FTCS '96: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, page 304, Washington, DC, USA, 1996. IEEE Computer Society.

[Cin07] Giancarlo Cinque. Valutazione della robustezza di un middleware per il controllo del traffico aereo mediante tecniche di iniezione dei guasti. Master's thesis, University of Naples Federico II, 2007.

[CTA02] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Trans. Computers*, 51(5):561–580, 2002.

[CYC+01] Andy Chou, Jun-Feng Yang, Ben Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In Greg Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM*

SIGOPS Operating Systems Review, pages 73–88, New York, October 21–24 2001. ACM Press.

[DK02] John DeVale and Philip Koopman. Robust software – no more excuses. In *Proceedings 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 145–154, (Bethesda, MD) Washington, DC, USA, June 2002. IEEE Computer Society.

[DKG99] John DeVale, Philip Koopman, and David Guttendorf. The Ballista software robustness testing service. Proceedings of TCS'99, Washington DC, 1999.

[DM02] João Durães and Henrique Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *PRDC '02: Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing*, page 201, Washington, DC, USA, 2002. IEEE Computer Society.

[DM04] João Durães and Henrique Madeira. Generic faultloads based on software faults for dependability benchmarking. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 285, Washington, DC, USA, 2004. IEEE Computer Society.

[DM06] João Durães and Henrique Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Software Eng*, 32(11):849–867, 2006.

[GKI04] Weining Gu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors. In *Proceedings 2004 International Conference on Dependable Systems and Networks (DSN 2004), Performance and Dependability Symposium*, pages 887–, Florence, Italy, June-July 2004. IEEE Computer Society.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[HBG+07] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *DSN*, pages 41–50. IEEE Computer Society, 2007.

[HL07] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *Operating Systems Review*, 41(2):37–49, 2007.

[IR85] R. K. Iyer and D. J. Rosetti. Effect of system workload on operating system reliability: A study on IBM 3081. *IEEE Transactions on Software Engineering*, 11(12):1438–1448, 1985.

[JAC+02] Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun, and Thomas Marteau. Analysis of the effects of real and injected software faults. In *PRDC '02: Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing*, page 51, Washington, DC, USA, 2002. IEEE Computer Society.

[JS05] Andréas Johansson and Neeraj Suri. Error propagation profiling of operating systems. In *DSN*, pages 86–95. IEEE Computer Society, 2005.

[JSM07] Andreas Johansson, Neeraj Suri, and Brendan Murphy. On the selection of error model(s) for OS robustness evaluation. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 502–511, Washington, DC, USA, 2007. IEEE Computer Society.

[KJA+04] Ali Kalakech, Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. Benchmarking operating system dependability: Windows 2000 as a case study. In *PRDC '04: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, pages 261–270, Washington, DC, USA, 2004. IEEE Computer Society.

[LI93] Inhwan Lee and Ravishankar K. Iyer. Faults, symptoms, and software fault tolerance in the tandem GUARDIAN90 operating system. In *FTCS*, pages 20–29, 1993.

[LRAL04] Jean-Claude Laprie, Brian Randell, Algirdas Avizienis, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, pages 17–30, 2004.

[MA89] John D. Musa and A. Frank Ackerman. Quantifying software validation: When to stop testing? *IEEE Softw.*, 6(3):19–27, 1989.

[MBD+06] R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira. Injection of faults at component interfaces and inside the component code: are they equivalent? *edcc*, 0:53–64, 2006.

[MDB+07] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental risk assessment and comparison using software fault injection. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 512–521, Washington, DC, USA, 2007. IEEE Computer Society.

[MKA+01] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johanson, and R. Lindström. Dependability benchmarking project deliverables. Technical report, 2001.

[MN07] Manuel Mendonça and Nuno Neves. Robustness testing of the Windows DDK. In *DSN*, pages 554–564. IEEE Computer Society, 2007.

[MP05] Leonardo Mariani and Mauro Pezzè. Behavior Capture and Test: Automated analysis of component integration. In *ICECCS*, pages 292–301. IEEE Computer Society, 2005.

[Mus93] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.

[PKS$^+$01] Jiantao Pan, Philip Koopman, Daniel P. Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness testing and hardening of CORBA ORB implementations. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS)*, pages 141–150, Goteborg, Sweden, July 2001. IEEE Computer Society.

[RM01] Laurent Reveillere and Gilles Muller. Improving driver robustness: An evaluation of the Devil approach. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS)*, pages 131–140, Goteborg, Sweden, July 2001. IEEE Computer Society.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222, New York, NY, USA, 2003. ACM Press.

[SC91] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, pages 2–9, 1991.

[SCK04] Daniel P. Siewiorek, Ram Chillarege, and Zbigniew T. Kalbarczyk. Reflections on industry trends and experimental research in dependability. *IEEE Trans. Dependable Secur. Comput.*, 1(2):109–127, 2004.

[SK05] Cristina Simache and Mohamed Kaaniche. Availability assessment of SunOS/Solaris Unix systems based on Syslogd and wtmpx log files: A case study. In *PRDC '05: Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[SKS02] Cristina Simache, Mohamed Kaâniche, and Ayda Saidane. Event log based dependability analysis of Windows NT and 2K systems. In *PRDC '02: Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing*, page 311, Washington, DC, USA, 2002. IEEE Computer Society.

[THB06] A.S. Tanenbaum, J.N. Herder, and H. Bos. Can we make operating systems reliable and secure? *COMPUTER: IEEE Computer*, 39, 2006.

[XKI99] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Networked Windows NT system Field Failure Data Analysis. In *PRDC '99: Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, page 178, Washington, DC, USA, 1999. IEEE Computer Society.

References from books

[BC05] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 2005.

[Bes05] Steve Best. *Linux Debugging and Performance Tuning: Tips and Techniques*. Prentice Hall, 2005.

[Dik06] Jeff Dike. *User Mode Linux*. Prentice Hall, 2006.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[Lyu95] Michael R. Lyu, editor. *Software Fault Tolerance*. Trends in Software. John Wiley & Sons, 1995. `http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html`.

[Lyu96] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996. `http://www.cse.cuhk.edu.hk/~lyu/book/reliability/index.html`.

[PP01] Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, 2001.

[PS01] John G. Proakis and Masoud Salehi. *Communication Systems Engineering*. Prentice Hall, 2001.

[RSCS02] Stefano Russo, Carlo Savy, Domenico Cotroneo, and Antonio Sergio. *Introduzione a CORBA*. McGraw-Hill, 2002.

Web resources

[1] CARDAMOM. `http://cardamom.objectweb.org`.

[2] Crash Core Analysis Suite. `http://www.missioncriticallinux.com/projects/crash/`.

[3] Direct Jump Probe. `http://lkst.sourceforge.net/djprobe.html`.

[4] Diskdump. `http://sourceforge.net/projects/lkdump/`.

[5] GDB. `http://www.gnu.org/software/gdb/`.

[6] Gprof. `http://sourceware.org/binutils/`.

[7] Kernel source tree, Documentation/fault-injection/fault-injection.txt. `http://www.mjmwired.net/kernel/Documentation/fault-injection/fault-injection.txt`.

[8] Kernel source tree, Documentation/kdump.txt. `http://www.mjmwired.net/kernel/Documentation/kdump/kdump.txt`.

[9] Kernel source tree, Documentation/kprobes.txt. `http://www.mjmwired.net/kernel/Documentation/kprobes.txt`.

[10] Linux Kernel Crash Dump. `http://lkcd.sourceforge.net`.

[11] Linux Kernel State Tracer. http://lkst.sourceforge.net/.

[12] Linux Test Project. http://ltp.sourceforge.net.

[13] LTTng and LTTV. http://ltt.polymtl.ca/.

[14] Mini Kernel Dump. http://sourceforge.net/projects/mkdump/.

[15] Minix. http://www.minix3.org.

[16] Netdump. http://www.redhat.com/support/wpapers/redhat/netdump/.

[17] Netkit. http://www.netkit.org.

[18] OProfile. http://oprofile.sourceforge.net/.

[19] SystemTap. http://sourceware.org/systemtap/.

[20] User Mode Linux. http://user-mode-linux.sourceforge.net.

[21] Virtualization. http://en.wikipedia.org/wiki/Virtualization.

[22] VMware Workstation. http://www.vmware.com/products/ws/.

[23] Xen. http://www.xensource.com.

[24] S. Best. Mastering Linux debugging techniques, 2002. http://www-128.ibm.com/developerworks/linux/library/l-debug/.

[25] W. Cohen. Gaining insight into the Linux kernel with Kprobes, 2005. http://www.redhat.com/magazine/005mar05/features/kprobes/.

[26] W. Cohen. Instrumenting the Linux kernel with SystemTap, 2005. http://www.redhat.com/magazine/011sep05/features/systemtap/.

[27] J. Corbet. Kexec, 2002. http://lwn.net/Articles/15468/.

[28] J. Corbet. Crash dumps with kexec, 2004. http://lwn.net/Articles/108595/.

[29] J. Corbet. Injecting faults into the kernel, 2006. http://lwn.net/Articles/209257/.

[30] S. Goswami. An introduction to Kprobes, 2005. http://lwn.net/Articles/132196/.

[31] H. Nellitheertha. Reboot Linux faster using kexec, 2004. http://www-128.ibm.com/developerworks/linux/library/l-kexec.html.

[32] P. Padala. Playing with ptrace, part i, 2002. http://www.linuxjournal.com/article/6100.

[33] P. Padala. Playing with ptrace, part ii, 2002. http://www.linuxjournal.com/article/6210.

[34] P. Panchamukhi. Smashing performance with OProfile, 2003. http://www-128.ibm.com/developerworks/linux/library/l-oprof.html.

[35] P. Panchamukhi. Kernel debugging with Kprobes, 2004. http://www-128.ibm.com/developerworks/library/l-kprobes.html.

[36] H. N. V. Goyal, E. W. Biederman. Kdump, a kexec-based kernel crash dumping mechanism, 2005. http://lse.sourceforge.net/kdump/documentation/ols2oo5-kdump-paper.pdf.